2

AD-A253 891

RL-TR-92-79
Final Technical Report
May 1992

SOFTWARE QUALITY METHODOLOGY INTEGRATION STUDY RESULTS

Rochester Institute of Technology

Jeffrey A. Lasky, Michael J. Lutz

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

RL-TR-92-79 has been reviewed and is approved for publication.

APPROVED: Roger J Dziegiel, Jr

ROGER J. DZIEGIEL, JR.
Project Engineer

FOR THE COMMANDER: [illegible]

JOHN A. GRANIERO
Chief Scientist
Command, Control, & Communications Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE May 1992	3. REPORT TYPE AND DATES COVERED Final Sep 87 - Jul 90

4. TITLE AND SUBTITLE SOFTWARE QUALITY METHODOLOGY INTEGRATION STUDY RESULTS	5. FUNDING NUMBERS C - F30602-88-D-0026, Task B-9-3333 PE - 62702F PR - 5581 TA - 20 WU - P9
6. AUTHOR(S) Jeffrey A. Lasky, Michael J. Lutz	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Rochester Institute of Technology 1 Lomb Drive Rochester NY 14613-5700	8. PERFORMING ORGANIZATION REPORT NUMBER N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory (C3CB) Griffiss AFB NY 13441-5700	10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-92-79

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Roger J. Dziegiel, Jr./C3CB/(315) 330-2054

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The thrust of this effort was to examine the software quality methodology and determine what improvements were necessary to improve utility of this technology. Rome Laboratory has been working in the area of software quality since the earlier 70's. The goal of the framework was to provide program managers a quantitative method for gaining insight into the quality of their software products (i.e. software requirements specification, preliminary design, detailed design, coding). This effort identified short falls in the methodology due to technology advances and the need to make enhancements.

Safety is one area the software quality framework does not support or address. This effort identified new quality factors that would be needed to address safety and software quality issues. Factors, criteria definitions, and metric element questions from the software quality model were reviewed within the context of defining safety.

Object oriented technology (OOT) was also examined to determine what should be added to the framework. New software developments are using object oriented design and to take advantage of this technology the framework needs to be updated. Seven software quality factors and ten software criteria were identified as being impacted by the use of object oriented technology.

14. SUBJECT TERMS Software Quality, Software Safety, Metrics, Object Oriented Technology	15. NUMBER OF PAGES 116
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

TABLE OF CONTENTS

SECTION I

EXECUTIVE SUMMARY

1. INTRODUCTION

This technical report presents the results of the Software Quality Methodology Integration Study. This work was performed for Rome Laboratory (RL) by Rochester Institute of Technology (RIT) under contract no. F30602-88-D-0026.

2. BACKGROUND

This effort continues work on the Software Quality Measurement methodology was has been under development by Rome Laboratory (formerly Rome Air Development Center) since 1976. Past RL sponsored work has been aimed at reviewing the Methodology as set forth in a three volume set of guidebooks developed by Boeing Aerospace Company and published in early 1985 [1]. A series of three guidebook validation studies were undertaken in the 1985-1987 time period. The purpose of these projects was to assess the feasibility and utility of transitioning the software quality measurement methodology to the software acquisition environment [2,3,4]. Each of these validation projects generated numerous recommendations for modifications to the methodology and to the guidebooks. In addition, a Software Quality Methodology Working Group was established, with representatives from contractor's who had ongoing efforts in the software quality measurement program. The working group met throughout 1988 and a compilation of the group's deliberations is found in [5].

More recent Rome Laboratory sponsored software quality efforts have focused on extensions and enhancements to the Methodology and on preparing the Methodology for transition into acquisition management practice. One of these efforts developed a cross reference guide between the Methodology's measurement elements and the corresponding data source location within the DOD-STD-2167A documentation structure [6]. This study also provided a method to integrate the software quality specification process within the Air Force's risk management framework [7], and a method to incorporate the Air Force's Management Indicators into the Methodology. Additional recent work includes completion of the first versions of the Methodology's two support tools, the Assistant for the Specification of Software Quality (ASQS) and the Quality Evaluation System (QUES). ASQS is an expert system designed to make available the Methodology's software quality specification process to acquisition managers who may not be experts in software quality technology [8]. QUES is a software quality system designed to support the data and calculation intensive components of the Methodology [9].

3. PRESENT STUDY

The objective of the present study is to support the continued evolution of the methodology. One of our study's two principal aims was to sketch out a research architecture to suggest future research activities for RL's software quality program. In order to preserve program continuity, we decided to again review the Methodology and the findings of the prior validation studies. This review was approached from a higher level perspective compared to our previous effort [6]. We also examined the Methodology's two support tools.

Our other principal aim was to support RL's intention to periodically update the Methodology in order to be consistent with revisions to DOD-STD-2167 and to incorporate advances in software engineering technology. The methodology was defined at the time DoD-STD-2167 was in effect. The current standard is DoD-STD-2167A which requires that a contractor perform a safety analysis if the software component of a system could contribute to placing the system into a hazardous condition. Thus, our study includes initial work on the definition of a fourteenth software quality factor *SAFETY*.

In order to support the objective of having the Methodology reflect current software engineering practice, we selected object-oriented technology for a second initial investigation. Interest and use in object-oriented technologies is rapidly accelerating and many of the claimed advantages of object orientation are directly related to software quality.

4. OVERVIEW OF CONTRACT RESULTS

4.1 Organization of the Report

Due to the wide scope of the investigations conducted under this contract, we decided to organize this report as a series of three separate studies, each with its own introduction and set of references. None of the studies makes reference to any of the other studies. Following are summaries of the contract's results.

4.2 *Software Quality Measurement Methodology*

The summary of our findings related to the review of the methodology is organized around three major observations:

1. *The Software Quality Methodology suffers from a lack of conceptual integrity.*

   Virtually every component of the Methodology is open to criticism due to the presence of arbitrariness or subjectivity. The net effect of this lack of conceptual integrity is a reduced degree of confidence in the Methodology and an impairment to acceptance and use.

2. *The Software Quality Framework is not evolving at a rate sufficient to adequately reflect the evolution of software engineering practice and DoD software strategy.*

   In order to be viewed as useful, the Software Quality Framework must be continually revised in a timely manner to reflect actual contractor software development practice. In the contracting community, software development practice is strongly impacted not only by DoD mandates (e.g., Ada) but also by DoD stated areas of emphasis and strategy (e.g., reuse). The Framework is lagging in this respect. Timely attention must be given to expanding the Framework's definition of the quality factor REUSABILITY to reflect the software quality impacts of reusing already existing software. We recommend that a periodic formal review of the SQF be undertaken in order to identify new and emerging practices, mandates and directions which will require SQF revisions.

3. *The specification of software quality should be based on mission area analysis.*

The satellite mission area analysis demonstrated the value and necessity of this type of software quality analysis [10]. The types of relationships found between satellite operational requirements and software quality factors underscore the value of mission area specific knowledge. Although expensive, difficult and time-consumming, detailed software quality mission area analyses will substantially improve the usefulness and acceptance of the Methodology. Such analyses will also provide the information required to generate mission area specific versions of the Framework. In its present form, the Framework is too abstract for use by individuals with operational concerns. Mission area specific versions should, at a minimum, use terminology specific to the mission area. An initial starting point would be to produce mission area specific guidebooks. Future development would concentrate on the creation of mission area Frameworks.

Based on our review of the Methodology's support tools, we recommended that no further development work be undertaken on the ASQS.

Finally, we sketched out a software quality research architecture which was suggested by our study. The resultant capabilities are intended to define a second generation SQM (Software Quality Maximizer) which emphasizes quality specification and method-specific measurement. With the exception of modest modifications to QUES, each architectural component represents a substantial research effort. The architecture is principally composed of:

1. Deep software quality mission area analyses
2. Development of ESSQU (Experimental System for Software Quality)
3. Development of QUMAX (Quality Maximizer), a support tool for conducting technical and cost tradeoff analysis
4. QUCOST (Quality Cost), a support tool for determining cost of quality
5. Definition of QUSPECS (Quality Method Specifications) which encapsulate key quality factor parameters.
6. QUES (Quality Evaluation System).

4.3 Safety and Software Quality

We undertook the following approach in this initial effort to include SAFETY as the fourteenth software quality factor. First, background reading was conducted in the area of software safety. Second, two important military standards, one each from the US (MIL-STD-882B Notice 1) and from the UK (MoD 00-55/1) that are centrally concerned with software safety were reviewed in detail. Third, the factor, criteria definitions, and metric element questions from the software quality model were reviewed within the context of defining Safety.

At this point, a determination was made that considerations of software safety within the context of the Methodology required the addition of two new criteria for the VERIFIABILITY factor. *Analyzability* refers to those characteristics of software which allow or facilitate rigorous analysis. *Predictability* refers to those characteristics of software which provide assurance that execution results conform to requirements.

Finally, a candidate set of sample metric element questions was created.

4.4 Object-oriented Technology and Software Quality

We undertook the following approach in this initial effort to expand the technological base of the Framework to include object-oriented technology. First, we identified the key concepts and assumptions underlying object-oriented software design and development methods. Second, we identified in the Framework seven software quality factors and 10 software criteria which we believe are strongly or moderately impacted by the use of object-oriented technology.

The majority of the selected factors are found in the Framework's Design group (Correctness, Maintainability) and Adaptation group (Expandability, Flexibility, Interoperability, Reusability). Only one factor (Usability) was selected from the performance group. The impact of object-oriented technology on each factor (and on their supporting criteria) is discussed.

Finally, a candidate set of sample metric element questions was created.

5. SUMMARY

This effort has produced a near and medium-term research agenda for Rome Laboratory's software quality research program. We undertook a comprehensive review of the RL Software Quality Measurement Methodology and of the Methodology's supporting tools so that our suggested research agenda would maintain continuity with prior work. During the course of our work, we concluded that the Methodology suffers from a lack of conceptual integrity. We believe that this condition, unless corrected, will substantially impair the acceptance and use of the Methodology. We also noted that the Methodology is evolving too slowly relative to the rates of change observable in software engineering practice. Our contribution in this later regard rests in the areas of software safety and object-oriented technology.

6. REFERENCES

[1] T.P. Bowen , G. B. Wigle, and J. T. Tsai, "Specification of Software Quality Attributes", Volumes I, II, and III, RADC-TR-85-37, February, 1985.

[2] James L. Warthman. "Software Quality Measurement Demonstration Project I", RADC-TR-87-247, December, 1987.

[3] Patricia Pierce, Richard Hartley, and Sullen Worrells, "Software Quality Measurement Demonstration Project II", RADC-TR-87-164, October, 1987.

[4] Dynamics Research Corporation, "Software Quality Guidebook Validation Results", Contract No. F19628-84-D-0016, Task 25, CDRLS 104-107, Task 73, CDRLS 104-110, September 30, 1986.

[5] Scientific Applications International Corporation and Software Productivity Solutions, Inc., "Software Quality Framework Issues", TR (Interim) Volume II, 2 June 1989, prepared for Rome Air development Center.

[6] Jeffrey A. Lasky, Alan R. Kaminsky, and Wade Boaz, "Software Quality Measurement Methodology Enhancements Study Results", RADC-TR-89-317, January 1990.

[7] AFSC Pamphlet 800-45. Software Risk Management. June, 1987.

[8] Larry Kahn and Steve Keller, "The Assistant for Specifying the Quality of Software (ASQS) Operational Concept Document, RADC-TR-90-195 Vol I (of two), Sept, 1990.

[9] Software Productivity Solutions, Inc. "Software Requirements Specification for the Quality Evaluation System (QUES), Contract No. F30602-88-C-0019, CDRL A002, September 5, 1989.

[10] Douglas Schaus, "Assistant for Specifying the Quality of Software (ASQS) Mission Area Analysis", RADC-TR-90-348, December 1990.

SECTION II

SOFTWARE QUALITY MEASUREMENT METHODOLOGY

1. INTRODUCTION

1.1 Specification and Measurement of Software Product Quality

The production of high quality software has become a visible national priority, both in the private and public sectors [1,2,3]. Software quality can be defined as a set of partially conflicting technical objectives which need to be explicitly incorporated into software system requirements and design. Examples of software quality objectives are reliability, maintainability, and usability. Historically, these quality objectives, often categorized as non-functional requirements, have been largely ignored, with virtually all attention being directed to producing software which meets functional requirements. The result has been near universal dissatisfaction with the operational (both functional and non-functional) characteristics of newly developed software. This dissatisfaction, coupled with unacceptable levels of software development costs and schedule, is generally known as the software crisis. Since virtually all studies conclude that future systems will have software requirements more complex than current ones, it is likely that software quality issues will continue to grow in importance.

Initial work in software quality specification and evaluation was conducted within the defense contracting industry. From the beginning, the objective was to develop a quantitative approach to quality specification and evaluation. The government's (the Air Force in this case) interest was to make available to acquisition managers an objective quality methodology which would support the imposition of measurable, contractually binding software quality requirements on contractors. The contractor's interest, which developed separately, was to have a workable and objective methodology which would help them improve the quality of delivered software systems. The earliest expression of a broad approach to quantitative software quality specification appears to be Boehm's work at TRW reported in a 1976 paper [4].

1.2 Specification and Measurement of Software Process Quality

For the past five years, the Software Engineering Institute has been refining a software process improvement model [5]. The impetus for the model's development was an Air Force request for a method of evaluating contractor's software development capabilities. The model defines five levels of increasing software development maturity. Each maturity level is associated with increased software development capability. Maturity is *maturity of the software development process*. The hypothesis underlying this model is that the level of achieved software product quality is determined by the quality of the software development process which produced the software product.

We believe, as do others [6], that the intensity of recent research shifts away from software product quality and towards software process quality is ill-advised. It would seem more natural to integrate the two related but differing viewpoints of how to improve software product quality. The key question to be addressed is to determine the *specific* relationship(s) between improvements in software process quality and the resultant improvements in software product quality (e.g., to what degree is increased software product reliability or maintainability due to software process improvements?)

1.3 History of Rome Laboratory Software Quality Research Program

Since 1976, Rome Laboratory (previously Rome Air Development Center) has been funding the development of the most comprehensive government program in software quality measurement methodology[1]. The objective of the methodology has been to provide acquisition managers with a method to quantitatively specify elements of software quality to be designed into a system and to measure the levels of achieved software quality when development is completed. The methodology also contains mechanisms for monitoring the achievement of software quality goals during the software development cycle.

1.3.1 Guidebook development

The RL software quality program has progressed through several stages. During the period 1976-1985, basic work in establishing and refining software quality concepts was undertaken. This first period culminated with the 1985 publication of a three volume set of guidebooks, *Specification of Software Quality Attributes* [13], which defined a full software quality methodology oriented towards Air Force acquisition managers. The guidebooks were produced by Boeing Aerospace Company.

The methodology can be viewed as having two major components. The structural component defines a software quality model and the relationships which exist among and between the structural elements. This structural component is the Software Quality Framework (SQF). The procedural component defines how the SQF is used to improve the quality of delivered software. The SQF, together with the procedural component defines the Software Quality Methodology (SQM).

Volume I of the guidebooks describes how the methodology can be integrated into the Air Force software acquisition management process. This volume also includes Boeing's enhancements to an earlier version of the underlying conceptual model and importantly outlines a suggested overall specification methodology. Volume II describes in detail the full software quality specification methodology. Volume III presents the quality evaluation procedures which yield quantitative indications of achieved software quality for both intermediate and final software products. The guidebooks were written to conform with DOD-STD-2167.

1.3.2 Validation studies

Since Volumes II and III were initial conceptual works, RL funded a series of studies during 1985-1987 to validate the methodology with actual project data. Unfortunately, opportunities to apply the methodology to ongoing projects did not arise. Instead, the methodology was applied *retroactively* to four prototype decision aids which comprised the Senior Battle Staff Decision Aids (SBSDA) system. The aids were designed to

[1]The RADC work has been the starting point for several other software quality measurement efforts, most notably ESPRIT's COQUAMO project [7, 8, 9, ,10, 11] and the IEEE Standard on Software Quality measurement [12].

demonstrate that artificial intelligence, decision analysis, and operations research techniques can assist senior AF managers in tactical battle situations. Each of two contractors, Computer Solutions, Inc. (CSI) and Scientific Applications International Corporation (SAIC) were assigned two decision aids. The contractors worked in parallel time frames but independently of each other. These two studies were named Demonstration Projects I [14] and II [15].

A third retroactive study was undertaken by Dynamics Research Corporation (DRC)[16]. In this study, the SQM was applied to a segment of the WWMCCS (World Wide Military Command and Control System) Information System (WIS). One objective of the WIS study not included in the SBSDA studies was to tailor the methodology specifically to Ada.

Although not a validation study, an additional review and analysis was conducted during 1988 by a Software Quality Framework Working Group [17]. The individual members of the group were primarily RL contractors who had recently worked on SQM related contracts. Several of the members had participated in SQM validation studies. This study was managed for RL by SAIC.

1.3.3 Support tool development

Early in SQM development, it was recognized that support tools would be required, since the total amount of data required by use of the methodology was considerable, especially at the unit source code analysis level. The first tool, whose development began in 1981, was a data collection/database system with report generation capability and a FORTRAN source code analyzer to automate measurement at the source code level.

Over time, the system concept broadened somewhat and the latest version the Quality Evaluation System (QUES) was developed by Software Productivity Solutions, Inc [20]. QUES is a database system which contains facilities for (1) defining a tailored framework, (2) providing forms generation and data collection facilities, (3) automatically conducting quality-oriented source code analysis for Ada, Ada PDL and FORTRAN 77, and (4) generating software quality evaluation reports.

At the time when the first validation studies were being started, RL also initiated an effort to develop a tool to facilitate use of SQM by AF acquisition managers. Air Force acquisition managers often do not possess in depth, application-specific software quality knowledge. So, the Assistant for Specification of Quality Software (ASQS) was developed by Dynamics Research Corporation to provide acquisition managers with access to software quality expertise. ASQS, (pronounced *asks*) is an expert system whose domain is the Volume II software quality specification procedures [18, 19]. Another reason for developing ASQS is that the methodology's technical and cost feasibility reviews become complex for large development projects and the ASQS provides the acquisition manager with reasoning and computational support to ease the task of trade-off analyses. The output from ASQS is a tailored version of the Software Quality Framework which is exported to QUES.

These tools are discussed in more detail in section 4 below.

1.3.4 Transition into practice

More recent RL funded activities have focused on transition of the methodology into (Air Force) practice. A study conducted by Rochester Institute of Technology (RIT) created a SQM/DOD-STD-2167A cross reference to aid in locating relevant SQM data inputs within the 2167A documentation structure [21] . The RIT study also provided frameworks for integrating the methodology within the Air Force's risk management model (AFSC 800-45), and for integrating the methodology and the AFSC Management Indicators.

Advanced Technology, Inc. (ATI) conducted an initial mission area analysis of five Air Force mission areas [22]. For two of the mission areas, domain-specific software quality knowledge was encoded into expert system rules and added to the ASQS rule base.

In 1991, RL created a Software Quality Technology Transfer Consortium organized under Cooperative Research and Development Agreements (CRDAs). The initial objective in forming the consortium was to solve the long-standing problem of validating the SQM within an environment of ongoing mission critical projects. Initial focus is on validating the reliability and maintainability quality factors. The consortium is viewed as the initial step in transition of the methodology into widespread practice.

1.3.5 The present study

The present study is RIT's second RL contract related to the Software Quality Methodology. Our two contract's have shared the same general objective of devising and/or recommending improvements to the methodology. We structured the present study into two tasks. Task 1 is to review the results of the software quality research program since the publication of the guidebooks (1985 to present), and based on this review (1) make recommendations for facilitating the transition of the methodology into practice, and (2) make recommendations for future software quality methodology research.

The methodology has already received considerable critical analysis [7, 14, 15, 16, 17, 21]. Our intent here is to focus on large issues which, if left unresolved, will likely impede the progress of the methodology's development and impair the rate of adoption by Air Force acquisition managers.

All of the validation studies referenced above reported numerous and substantial problematical issues resulting from the attempted use of the methodology. Nonetheless, each concluded, and in our opinion without justification, that the methodology is valid and usable. We feel that the Guidebooks which outline the methodology should have been considered as a first attempt to define a SQM, rather than as a completed SQM development. The careful reader will detect a tension in our report between the recommendations for improving the current state-of-affairs and an uneasiness about the ultimate usefulness of the SQM.

Task 2 recognizes that the software development arena continues to undergo change since the 1985 publication of the SQM and that the SQM needs to be continually revised so it remains current and useful. In light of this need, we selected two new developments in software engineering for initial study which are completely absent from the methodology. The first is software safety. Unlike the version of the DOD Software Development Standard in force during the Boeing work on the guidebooks, the current standard, DOD-STD-2167A, mandates a software safety analysis for systems were malfunctions could have serious consequences. Thus, we reviewed selected literature on software safety and defined a new software quality factor SAFETY together with two new software criteria and a candidate set of metric questions.

The second development is object-oriented design (OOD). OOD was not in wide spread use during the 1982-1985 guidebook creation period. (The same, of course, is true of the Ada language. Ada has raised the visibility of OOD within the contracting community). Since OOD is experiencing very rapid technology tra^nsition, we believed that the SQM should explicitly recognize the potential contributions of OOD to improved product software quality. Thus, we reviewed selected literature on OOD and defined a candidate set of OOD metric questions related to several criteria and to several factors.

The remainder of this report is organized as a set of descriptions about various components of the SQM. Each description is followed by an analysis section which usually also includes recommendations for improvement. Readers with more than a passing interest in the methodology should have read or have ready access to the Guidebooks [13], and a basic description of ASQS [18] and QUES [20].

2. SOFTWARE QUALITY FRAMEWORK

A definition of the software quality framework (SQF) is primarily structural and includes the software quality model, definition of quality factors and the number of discriminating factor goal levels, criterion, metrics, and metric elements, denotation of interrelationships among factors and between factors and criteria, and specification of factor impacts on cost of quality.

2.1 Software Quality Model

The key concept of the framework is a three level hierarchical model of software quality.

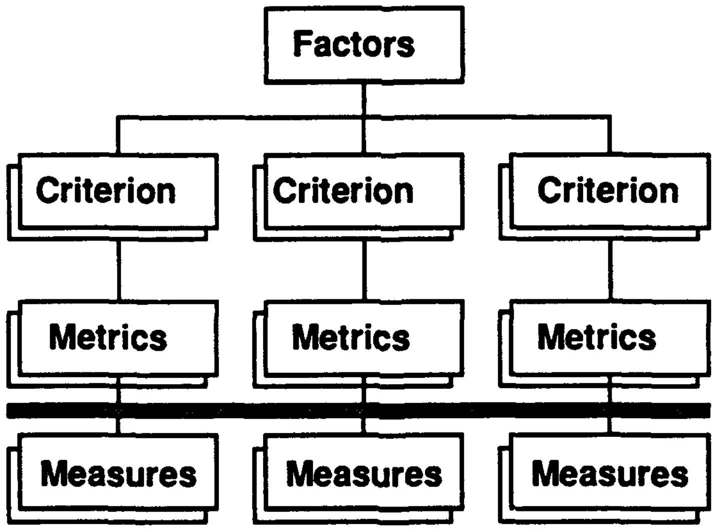


Figure 1. Software Quality Model. The level under the grey bar is usually not viewed as an additional level.

The top level is a set of thirteen customer-oriented *software quality factors*. The factor set is Efficiency, Integrity, Reliability, Survivability, Usability, Correctness, Maintainability, Verifiability, Expandability, Flexibility, Interoperability, Portability, and Reusability. The reader should keep in mind that the quality factors are not perfectly orthogonal; important relationships exist among the factor set.

In the current version of the SQF, three levels of factor goals are identified: excellent, good and average. For quantitative goal setting, excellent is mapped into .90 to 1.00, good into .80 to .89, and average into .70 to .79. The role of these factor goals in the overall methodology is considerable. For example, at the end of the development, a score is calculated, in the range of 0.0 to 1.0, which represents how much of a certain quality factor is actually present in the final product. That final score is then compared to the target goal for that factor to determine if the contractor has meet requirements. One goal of the RL Software Quality Technology Consortium is to determine the validity of this approach to software quality measurement.

The second level is a larger set (twenty-nine) of defining attributes for the software quality factors. These are termed *software quality criteria* and reflect technical (developer) considerations of good software engineering and development practice. For example, the RELIABILITY software quality factor is defined in terms of three software quality criteria: accuracy, anomaly management and simplicity. Some criteria are used in the definition of more than one factor. The reader should keep in mind that the quality criteria are not perfectly orthogonal; important relationships exist among the criteria set.

The third level is the measurement level. Here, seventy-three *metrics* are used to represent the degree to which software quality is present in intermediate and final software products. Measurements are scaled from zero to one, where a value of zero represents a total absence of some component of software quality and a value of one represents the highest achievable (within the context of the SQF) level of a component of software quality. Each metric is subdivided into one or more metric subsets. Each subset is concerned with a specific dimension of software quality. For example, the anomaly management metric is subdivided into seven metric subsets: error tolerance/control, improper input data, computational failures, hardware faults, device errors, communication errors and node/communication failures. In this example, each subset measures the degree to which the software can provide for continuity of operations given the occurrence of different types of errors and failures.

The metrics in turn are defined by lower level metric elements. The metric elements are the direct and atomic measures of software quality. There are approximately three hundred metric elements. In the SQM, most direct measures are acquired by the answers to binary (yes/no) questions. Since SQM is a quantitative model, yes answers are mapped to a value of one, no answers to a value of zero. For example, one data point used to assess the software's design with reference to computational failures is obtained from the following metric element question:

> Are all critical subscripts (supporting a mission-critical function) checked for out-of-range values before use?

2.2 Technical Interrelationships

The explicit recognition of technical interrelationships among factors and between factors and criteria in the SQF is unique among software quality models. It provides the basis for a systems engineering approach to specifying software quality requirements. The question to be answered is: given a set of quality factors each assigned a quality level goal, can these goals be reached? Or, put another way, is this a feasible goal set?

These technical relationships are either positive (beneficial) or negative (adverse). At the factor level, if factor X positively impacts factor Y, then the presence of factor X will increase the likelihood of achieving the desired quality goal for factor Y. If the indicated relationship is negative, then the presence of Factor X will increase the difficulty of achieving the desired quality goal for Factor Y. For example, their is a defined negative relationship between the factors Efficiency and Maintainability. That means that it will be difficult to achieve the target goals if both factors are assigned a level of Excellent (High). The rationale is that high hardware efficiency is often obtained by use of programming practices which impair future maintainability. The SQF also defines the relative strength of the interrelationships in terms of degree of impact. Not all pairs of factors are interrelated. If they were, a tradeoff resolution of factor conflicts would be virtually impossible. In many cases, the impact is one to many, so that factor Y above could also represent a set of factors. Similar reasoning applies to interrelationships between factors and criteria.

2.3 Analysis

2.3.1 Three level quality model

We note that none of the validation studies mentioned in section 1.3.2 above have questioned the three level structure of the SQF. On close examination, we believe that the presence of the middle (criteria) level is a major contributor to the overall complexity of the SQM, and therefore a barrier to statistical validation and a barrier to ultimate contractor acceptance. An important question to resolve is: what would be the consequences of eliminating the second level and transforming the SQF into a two level model?

As far as the acquisition manager is concerned, he or she is only concerned with the factor (top) level.

Analysis of factor interrelationships is made more difficult by the presence of a criteria level. The network of criteria relationships which directly underlie factor relationships must be understood and considered by whoever conducts the software quality specification procedure.

The quality factor evaluation scoring algorithms (simple combinations of criteria scores) can be modified by changing the weights associated with each criterion. There are two primary uses of criteria weighting in the SQM. One use is that weights conceptually allow certain elements of factors to be emphasized in terms of contributions to quality. The other use represents one suggested way of resolving factor interrelationship conflicts (for an example, see page 4-37 in Guidebook Volume II). The availability of criteria weighting impairs the task of validating the SQM, since another source of variability has (potentially) been introduced.

On the other hand, criteria weighting and criteria interrelationships may provide valuable design and tradeoff information to the development team.

Recommendation 1. Perform an in depth analysis of the consequences of retaining versus deleting the criteria level.

2.3.2 SQF definition

The three demonstration studies and the Working Group review reported numerous and substantial concerns with the basic SQF definition. These concerns were typically semantic issues as the various researchers struggled to make recommendations which would produce a more consistent, less ambiguous and comprehensive SQF definition. In light of almost universal concerns of a variety of kinds[2], the reluctance on the part of the SQM community to modify the Guidebook Volume II SQF definition is surprising.

The general rationale for this reluctance, as noted in the Working Group's report [17], was that since the existing definition had not yet been validated, modifications, especially those of an expansive type (e.g., add new factors) would only make matters worse. We are not in full agreement with this point of view.

It is important to keep in mind that the basic SQF definition task was started in 1982. During the intervening ten years, customers and developers have become substantially more aware of and more sophisticated about the multidimensional character of software development. In addition, technology has advanced. We believe that an out-of-date and ambiguous SQF definition will both limit the applicability and value of the SQM, and hinder acceptance by the contracting community.

As an example, consider the current SQF definition of the factor REUSABILITY: relative effort for converting a portion of [currently being developed] software for use in another application. This is the export view of reusability. However, the DoD and DARPA have stressed the prime importance of the import view of reusability. Much of the strategy outlined in the current DoD Master Software Plan relies on import reusability and related technical and managerial issues. The DoD/DARPA concept of megaprogramming, the construction of new software systems based to the greatest extent possible on existing software, clearly frames the issue. Contractors are being put under pressure to increase import reusability, but the current SQF definition completely ignores this emerging and high priority viewpoint. To the extent that import reusability is available on a given development project, the SQF distorts reality because it does not admit the positive quality impacts claimed for software reuse. The determination of the final factor goals which the contractor is being pressured to achieve might be very different if the REUSABILITY factor was split in export and import reusability.

Recommendation 2. Based on prior evaluations, DoD directions, and current technology, modify the Guidebook definition of the basic SQF.

2.3.3 Factor and criteria Interrelationships

One of the more valuable concepts in the SQF definition is the explicit recognition and quantification of software quality factor and criteria specification as a multiple criteria decision making problem with conflicting goals (although not named as such).

[2]We ignore the specific concerns which were raised due to reliance on manual data collection and analysis since the QUES tool provides substantial improvements in this area. It is also important to note that all of the demonstration project's systems developments were pre DOD-STD-2167. Each contractor, therefore, was required to expend considerable effort to translate the available documentation to 2167 terminology and structure. These activities should also be ignored since they are anomalies due to the timing of the demonstration projects and are not related to core structural or methodology issues.

Guidebook Volume II contains several tables and figures which depict, in tabular form, factor and criteria interrelationships, together with indications of degree of effect. The intent of these tables is provide the basic information required to perform a tradeoff analysis for the purpose of adjusting the quality factor goals to arrive at an achievable quality specification. We believe this is a unique feature and valuable of the RL SQF.

Rationales are found in Volume II for the specified structure and intensity of the interrelationships. However, at this point in time, the current validity of the interrelationships is open to debate. The Guidebooks may in part be reflecting 10 to 15 year old software development perspectives and technology. The mandated use of Ada in increasing numbers of DoD acquisitions offers a bounded and high-priority opportunity to examine the validity of the interrelationships, both in the current and future SQF versions.

Recommendation 3. Initiate an effort to build an Ada-based software quality testbed in order to experimentally determine factor and criteria interrelationships and their associated magnitudes.

2.3.4 Metric questions

In the current SQF definition, each metric has a varying number of metric elements mostly structured as YES/NO/NA (not applicable) questions. The metric elements represent a sample of review points asserted to be useful in determining to what extent specific aspects of quality are present in software. *The guidebooks offer no rationale or methodology for the selection of metric elements found in Guidebook Volume III.* The lack of such rationale hinders contractor acceptance (appearance of arbitrariness, lack of confidence in correctness of measurements) and impairs the future evolution of the framework.

Each NA response effectively reduces the sample size which determines the value of the metric and so reduces the metric's information content. The questions are the same independent of the target goals specified for the quality factors.

This approach makes little sense. It suggests that the level of software quality present in the product is identical for all same sized YES subsets of the metric elements for a given metric X. For example, suppose that metric X is defined by 10 metric elements. For all combinations of exactly four YES responses, the metric will receive a value of 0.4, independent of which combination of four questions received YES responses. The discriminating value of the metric is thus questionable.

A more discriminating metric would be obtained by having one set of questions for each possible factor level in the SQF definition. The set answered is determined by the level of the factor goal. This approach maximizes the information content of each metric, since the number of NA responses should be minimized.

Two implementations are possible. In both cases, three question sets are created. In one case, only the question set which corresponds to the factor goal level is used. In the other case, the question sets are cumulative, so that for example, all three question sets would be selected if the factor goal assigned was excellent. This approach will work even if the criteria to which the metric is attached is shared across two or more factors with differing factor goals. In that case, the highest factor goal level determines the maximum number of question sets to be used. Then, after that number of question sets are answered and scored, the number of question sets used to report values up the hierarchical model to the shared criteria is determined by the (differing) factor goal levels.

The total number of metric elements required for each metric would, in general, not increase in direct proportion to the number of distinct goal levels in the framework. This is because (1) the number of NAs would be less and (2) in the cumulative approach, fewer questions for each goal level would (might) be required.

Recommendation 4A. Develop a rationale and procedure for the selection of metric element questions

Recommendation 4B. Determine the cost/benefit of creating, for each metric, one set of metric element questions which reflect the technical requirements necessary to achieve each possible factor level.

3. SOFTWARE QUALITY METHODOLOGY

3.1 Determine Quality Goals

The Volume II quality specification methodology is designed to be used in the early stages of system definition. In brief, quality specification is performed as a four stage procedure. The first step in the software quality specification procedure is to identify system functions which will be supported by software. Although the guidebook contains an example functional decomposition for C^2 applications, *the guidebook provides no general guidance on how to decompose systems into functions.*

In the second step, the functions are reviewed in order to determine which quality factors are applicable A target quality goal is then allocated to each factor. In general, only a subset of the top level quality factors will be assigned to each function. Information needed to assign quality goals is fused from the system description (RFP, A-Spec), a quality survey of the customer and likely users, and any existing quality assignments from similar systems. At the end of the second step, then, an initial software quality goal specification is available. This specification is typically recorded in matrix form, where the value of each non-empty cell represents an initial quality goal for a {system function, quality} pair. Empty cells mean that a specific quality factor was not applicable or not important to a specific system function.

3.2 Determine Technical Feasibility

In the third and fourth steps, the initial quality goals are reviewed for technical and cost feasibility.

Factor and criteria relationships are used in the methodology to adjust the initial factor goal matrix until an achievable (feasible) set of factor goals is reached. Often, there is more than one set of adjustments which will lead to a feasible set. *The guidebook provides no guidance on how to revise the initial quality goals based on factor and criteria interrelationships.*

The adjusted factor goal matrix is then reviewed to determine the relative net cost/benefit of applying the SQM to a software development effort. Net cost includes the additional costs associated with specifying quality requirements, allocating those requirements to more detailed levels of requirements and design, designing and building quality into the product, and evaluating the achieved quality levels. Net benefits include increased awareness of quality throughout the life-cycle, early problem detection, fewer defects, reduced effort, and higher quality products.

Quality factors are considered within the context of how they impact costs; i.e., positively or negatively. Volume II contains suggested relative cost ranges for each factor's cost impact on each of four life cycle acquisition phases: demonstration and validation, full-scale development (Pre CDR), full-scale development (post CDR), and Production and Deployment. In addition, positive and negative factor interrelationships are also analyzed for cost/benefit impact. After the cost/benefit analysis has been completed, the guidebook states that quality goals should be reviewed for life-cycle cost considerations and revised if necessary. ***The guidebook provides no guidance on how to revise the initial quality goals based on cost considerations.***

Recommendation 5. Create the missing guidance sections and revise Volume II accordingly.

3.3 Measure Achieved Quality

The Volume III quality evaluation methodology is designed to be used both during software development for monitoring of software quality goal progress and at the end of development activities to assess achieved levels of software quality. Quantitative measures of software quality are calculated by an aggregate scoring algorithm. Values from the model's lowest level (metric elements) are averaged and aggregated up to the metric level. Metric values are averaged and aggregated up to the criteria level. Factor scores are then calculated by the weighted sum of that factor's defining criteria.

3.4 Analysis

3.4.1 Determine quality goals

A process of ad hoc identification of system functions via functional decomposition is too subjective and will introduce undesirable variance into the SQM. The same subjectivity is present in assigning factors and allocating quality goal levels.

One way of minimizing variance is to develop generic, domain specific quality specifications. (Volume II presents a generic functional decomposition for the Command and Control domain.) Although relevant quality factors are allocated to functions, no quality goal assignments are shown.

A set of initial generic functional decompositions for five Air Force mission areas (Armament, Avionics, C^3, and Missile/Space) was developed for RL by Advanced Technology, Inc. (ATI) [22]. The generic decompositions were part of a larger mission area analysis study aimed at identifying mission and system characteristics that impact software quality in the Intelligence function of C^3 and in the Satellite mission area. The software quality findings were expressed in the form of expert system rules and were later incorporated by into the ASQS support tool. With access to mission area specific software quality information, ASQS is then usable by an acquisition manager for assigning quality goals to factors.

Given the embryonic development stage of the ASQS's knowledge base, the ATI study cautioned against reliance at this time on using ASQS for assigning goals to factors.

We agree with this assessment and suggest that additional mission area domain analyses be performed in order to develop more detailed knowledge about the software quality requirements for these mission areas. The results of the domain analyses could be used to develop more detailed functional decompositions and to develop corresponding generic factor goal assignments.

Recommendation 6. Initiate additional mission area domain analyses.

3.4.2 Determine technical feasibility

The quality factor technical feasibility and tradeoff analysis is a problem in multiple criteria decision making with conflicting goals. This class of problem has generally resisted analytical solution. In the case of the methodology, the conflicting goals are derived from the factor interrelationships (at the surface level) and from the criteria relationships (at the deep level). The problem can be stated as:

Maximize the quality goals for a set of factors subject to resource constraints.

The methodology does not explicitly consider resource constraints. The cost/benefit analysis outlined in the guidebook answers the question: What is the net economic benefit of applying the methodology to a software development effort? While this is of clear interest, the output of the analysis is at the macro project level. Thus, the methodology's consider costs procedure does not directly support considerations of cost when determining feasible sets of factor goals.

We believe this is a serious weakness in the methodology. A systems engineering type analysis is required where factor and criteria interrelationships are considered simultaneously with cost constraints. With the addition of cost constraints, an acquisition manager can gain a deeper understanding of what factor goal levels are in fact reasonable to set as software requirements.

A systems engineering approach raises two concerns. First, the criteria/factor/criteria-factor relationships generate a substantial level of complexity. Second, costs related to achieving specific levels of quality factors are generally not available.

The complexity of quality factor/quality criteria interrelationships is apparent in figure 2, which is a representation of interrelationships for the FLEXIBILITY quality factor. This form of representation was developed by Pierce in the course of conducting one of the validation projects [15].

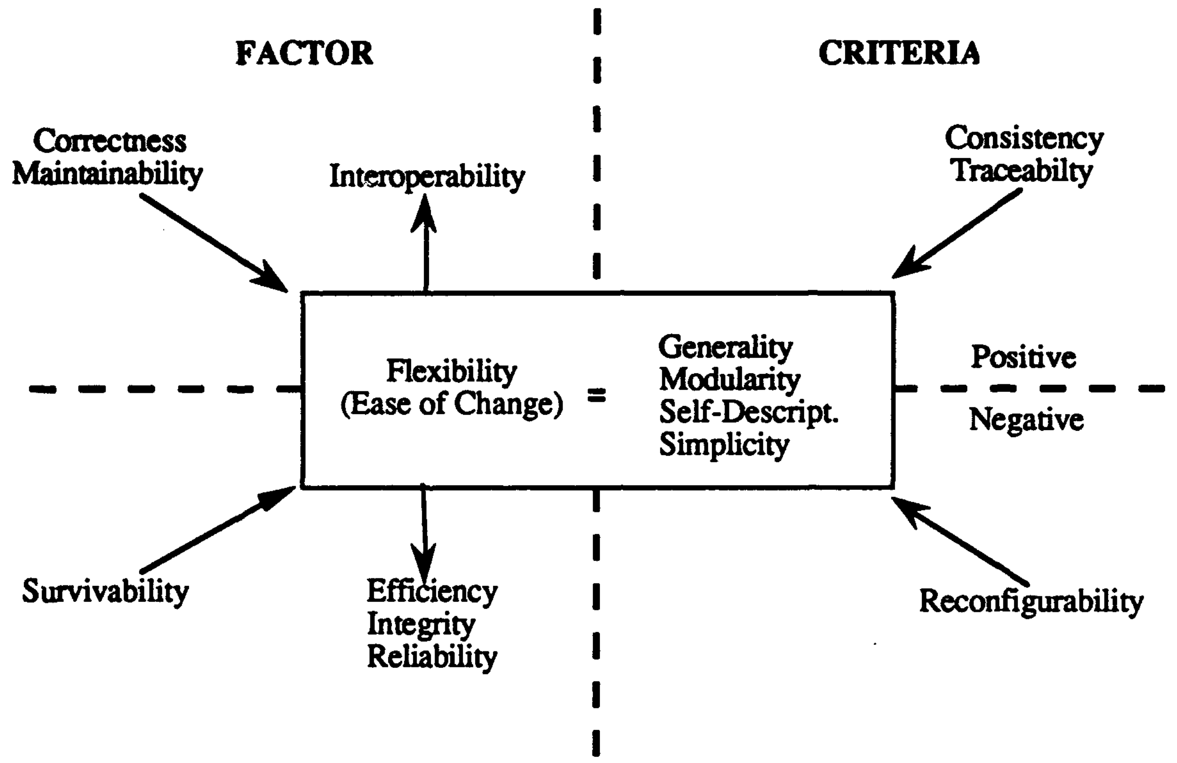


Figure 2. Factor/Criteria Interrelationships for FLEXIBILITY

Figure 2 depicts seven types of relationships:

1. <u>Positive factors impacting</u> FLEXIBILITY (Correctness, Maintainability)

2. <u>Negative factors impacting</u> FLEXIBILITY (Survivability)

3. <u>Positive criteria impacting</u> FLEXIBILITY (Consistency, Traceability)

4. <u>Negative criteria impacting</u> FLEXIBILITY (Reconfigurability)

5. FLEXIBILITY <u>positively impacting other factors</u> (Interoperability)

6. FLEXIBILITY <u>negatively impacting other factors</u> (Efficiency, Integrity, Reliability, Survivability)

7. <u>Criteria which define</u> FLEXIBILITY (Generality, Modularity, Self-Descript, Simplicity)

If one were to produce a set of connected graphs for the complete set of quality factors and quality criteria, it would be clear that summation over all criteria, factor and criteria/factor relationships produces a highly complex web of software quality interrelationships. Some type of computer-based support needs to be provided to assist individuals in dealing with this level of complexity for multi-criteria tradeoff problems.

Costs of software quality have been largely ignored in formal planning for software development efforts. Basic research needs to be performed in order to develop a database of software quality costs. Such cost data could be incorporated into the methodology to permit the type of tradeoff analysis described above to take place, to validate the guidebook cost ranges used in the methodology's cost/benefit analysis, and to form the empirical base for a COCOMO-like cost of quality model using the methodology's quality factors as the model's independent variables (quality drivers analogous to COCOMO's cost drivers.

Section 4.1.4 of Volume II contains Figures showing the approximate quality factor impacts on development costs over each of the four major acquisition phases. *The guidebook offer no rationale for the suggested cost percentage values.* The lack of such rationale hinders contractor acceptance (appearance of arbitrariness, lack of confidence in correctness of cost estimates) and impairs the future evolution of the framework.

Recommendation 7A. Initiate an effort to develop a tradeoff assistance tool which would provide support for resolving factor/criteria interrelationships subject to cost constraints.

Recommendation 7B. Initiate an effort to collect basic cost of quality data.

Recommendation 7C. Initiate an effort to develop a cost of quality model.

4. SUPPORT TOOLS

4.1 Assistant for Specifying the Quality of Software

The Assistant for Specifying the Quality of Software (ASQS) is an expert system whose principal objective is to facilitate the transition of the SQM into acquisition management practice. The underlying strategy to accomplish this objective was to provide facilities which partially automated the software quality specification process as outlined in Guidebook Volume II. The ASQS was developed by Dynamics Research Corporation during the period 1985-1990. In certain areas, DRC incorporated their interpretations and extensions to the specification process based on findings from their SQM validation contract. The current version of ASQS should be considered as a proof-of-concept demonstration.

The rationale which lead to the development of the ASQS was that system acquisition managers are typically unfamiliar with software quality concepts and technology. So, they would need assistance in translating their knowledge of software-intensive system characteristics and requirements into a software quality specification. In addition, the SQM technical and cost feasibility analyses would likely be sufficiently complex that computer support would be required for all but the simplest systems.

In order to support the first part of the quality specification process, identify functions and assign initial quality goals, the long range plan for the ASQS included parallel efforts to conduct domain analyses in Air Force mission areas. The domain analyses were to yield two categories of relevant information which would be added to the ASQS's knowledge base.

One category is generic functional decompositions. The first step in the quality specification process is identify functions. In the absence of a complete generic description, too much variability across systems in the same mission area would be experienced. Functional decompositions were developed for four major mission areas.

The other category was software quality analyses to determine the relationship between operational requirements of a system and corresponding software quality requirements. This information is used to support the generation of the initial function/factor goal matrix. Again, in the absence of such application-specific information, quality specifications would tend to be to subjective and incomplete. The initial attempt to conduct a software quality focused domain analysis yielded results (albeit incomplete) for the intelligence and satellite mission areas.

The ASQS strategy for assigning initial goals to factors is to accumulate evidence that a quality goal is implied from a variety of data sources. The data are fused using a heuristic algorithm which weights the data with degree of importance factors. The data sources include (a) system and software characteristics which may be implied from mission area and operational environment characteristics, (b) development characteristics (e.g., Ada), quality survey data from customers and users, (c) mission area analyses, and (d) application specific quality concerns which may be known to acquisition managers.

Another important source of data is the ASQS user. In order to acquire as much information and evidence as possible, the ASQS guides the user through a structured question-answer dialogue. The ASQS will only ask questions specific to the selected mission area and only for information which ASQS cannot infer from the other available sources and questions already answered. The sequence of questions is guided by the user's responses. Such a facility is generally regarded as a 'smart interface'.

After the initial goal matrix is complete, the ASQS conducts the technical feasibility analysis. The actual method used was developed by DRC and documented in [CDRL 5 of reference 16]. The ASQS user is presented with the initial goal matrix annotated to show ASQS suggestions for revision. If desired, the user can override the ASQS suggestions.

The last step in the specification process is to review the goal matrix for cost considerations. This analysis is limited to informing the user that, in some cases, there is sufficient evidence that an adverse factor relationship(s) exist. The implication is that it may be excessively costly to achieve specified factor goals. The final factor matrix is then produced.

At any time during the specification process, the user can ask WHY or HOW and receive explanations from ASQS.

ASQS uses the final goal matrix to generate a tailored version of the SQF for export to the QUES tool. The tailoring process consists of selecting all the criteria implied by the set of selected factors and selecting those metric questions (which define the criteria) most relevant to the final quality specification within the context of what the ASQS knew and/or was told about the system to be developed. Since the SQM is intended to be used for project monitoring (as well as final product quality assessment), the ASQS will generate a tailored version of the SQF for each development life-cycle phase.

Interested readers should consult the ASQS Operational Concept Document [18] and the ASQS User's Manual [19] for additional ASQS features.

4.1.1 Evaluation: core functionality

In our opinion, an expert system is not required to accomplish the transition of the SQM into acquisition management practice. First, an expert system such as ASQS needs *deep world* knowledge if it is going to perform adequately on real projects. Based on an examination of the ASQS rule base, we concluded that, at the present time, ASQS has acquired only surface world knowledge. The inferences that create the initial factor goal matrix are very elementary; i.e., ASQS is not very smart about factor goal allocation. Deep world knowledge about software quality requirements for specific mission areas is probably available only from the results of deep domain analyses. This is especially true if one wishes to allocate quality goals to the lower level (CSU) software designs.

Once deep domain analyses are available, what is to be gained from encoding the domain knowledge into an expert system? Generic domain specific functional decompositions together with factor goal assignments will then be available. Given the size of the domains being contemplated, and in many cases the substantial differences in system requirements and system architectures, continued development of the ASQS rule base must be considered a high risk development effort.

The impact of any desired changes on the generic quality specification could be assessed with the tradeoff assistance tool (recommendation 6A). There is no compelling reason no compelling need to use an expert system to reason about any potential ripple effects on the generic specification. While certain other functions of the ASQS might very well be worth preserving (e.g., smart tailoring of the SQF), it is just as likely that non-smart tailoring, together with an increasing number of individuals familiar with the SQM technology will suffice.

As far as the technical/cost feasibility process outlined in section 3.4.2 above, this clearly appears to be a computable problem not one especially well-suited for an expert system.

Recommendation 8. Stop development work on the current ASQS concept.

Recommendation 9. In the event that recommendation 8 is not accepted, determine the scope of the ASQS knowledge base which would be required to generate meaningful mission area specific factor goal assignments.

4.1.2 Evaluation: current implementation

In the event that our recommendation to stop development work on the ASQS is not accepted, we wish to stress the necessity of abandoning the current ASQS implementation as the baseline for future work. At the time when work was started on the ASQS, Lisp machines, such as the Xerox 1186, were still popular and the Xerox version of Lisp, Interlisp D, was considered an attractive Lisp development environment. EMYCIN, which is essentially the classic expert system MYCIN stripped of its domain knowledge, was and still is a popular expert system shell. However, as the price/performance levels of personal computers and workstations steadily improved, and expert system technology became popular, more contemporary and functional expert system development environments became available. Following are the rationale for our recommendation to abandon the current ASQS implementation. In the next section, we present our reasons for recommending Nexpert Object' as a future development system.

First, the Xerox 1186 is a dead platform (no longer marketed) with non-competitive performance. Second, the software is essentially non-transportable. Third, there are no built-in high-level graphic user interface generation facilities in Interlisp D. We found approximately 50% of the total ASQS Lisp code (and perhaps 50% of total development effort) dedicated to the graphic user interface. Fourth, the use of EMYCIN as the inference engine limits rule base representation, design and performance options. EMYCIN rule bases cannot be partitioned. So, as the rule base grows in size, ease of maintenance will decline rapidly. Use of the ASQS showed that performance on the current 380 rule system is only minimally acceptable. This is due to a lack of search control options in EMYCIN and the low performance of the Xerox 1186. Performance on an expanded base of thousands of rules would be unacceptable. Fifth, EMYCIN/Interlisp provides no meaningful high-level expert system development support (no graphical representation of the knowledge base, no graphical dynamic traces of program execution). Consequently, an ASQS with an expanded rule base of thousands of rules would be extremely difficult to validate. Sixth, none of the advantages of open systems architecture are available.

4.1.3 Evaluation: future implementation

Nexpert Object[3] is a contemporary, comprehensive development environment for expert systems and other knowledge-based applications. It is in wide-spread general use as well as in use by the military. For example, Nexpert Object is being used by Northrop in the development of the US Air Force's ATMCS system (Advanced Tooling Manufacture for Composite Structures). Following is an overview of Nexpert Object's advantages.

High level of portability.

Nexpert was developed in the C language with portability a major design goal. There is a version of Nexpert for virtually any of the platforms currently in wide spread use. Recently, Open Interface', a portable graphics user interface (GUI) toolkit was added to the product line. A GUI developed on one platform using Open Toolkit can be ported across different platforms and operating systems. Thus, entire Nexpert applications become portable.

Open Architecture.

Nexpert provides a bridge to several database systems (including Oracle) and to Macintosh's Hypercard. In addition, the Nexpert system architecture allows Nexpert to be the system's controlling software or allows Nexpert to be embedded within and controlled by another application. The later capability is provided by the Application Programmer's Interface. This interface supports both C and Ada.

Object orientation

This architecture feature provides the capability to separately structure the knowledge base and the rule base, thus facilitating evolution of both and also increasing comprehension of larger applications. All of the advantages of classification and inheritance are made available to the designer. In addition, the designer is given options to control the inheritance mechanisms and thereby improve performance.

Partitioning

Nexpert provides two mechanisms to group related rules and to define separate knowledge bases. A knowledge island is an implicit grouping of rules. The systems determines grouping by identifying rules with the same hypotheses or with common LHS (conditions) data. The value of this mechanism is that Nexpert's inferencing mechanisms attempt to fully process one knowledge island at a time. The provides the designer with a way to modularize the application at the rule level.

[3]Nexpert Object is marketed by Neuron Data, Inc., 156 University Avenue, Palo Alto, CA. (415)-321-4488.

A knowledge base is an explicit grouping of objects, classes, properties, rules and knowledge islands. Knowledge bases are loaded into memory as needed. This capability is especially valuable in larger applications which may have to process several relatively applications domains. For the ASQS, each mission area might be designed as a separate knowledge base. Application evolution is improved as well as processing performance.

Inferencing

In Nexpert, rules are symmetric. This means that the rules do not have to be designed for either forward chaining or backward chaining. Nexpert will select the appropriate search strategy based on the context of the problem. In addition, there are several search design options and a rule priority mechanism to improve performance. Nexpert also supports opportunistic and non-monotonic reasoning.

Development support

The rule and knowledge bases are entered on forms. Graphic facilities are available to visualize the structure of the rule and knowledge bases and also to support execution tracing of rule processing.

In order to take full advantage of Nexpert Object's capabilities, a Nexpert implementation of the ASQS would require a total redesign. However, the rule base is relatively small and GUI generation would be accomplished by either toolkit access to the host's native GUI functions or by use of the portable Open Interface toolkit.

4.2 Quality Evaluation System

QUES is an object-oriented database system which can generate hierarchical data structures useful in representing software quality definition and evaluation models. QUES is not limited in scope to the SQF; other arbitrary frameworks can be designed. Although precursors to QUES were designed solely as data collection and data evaluation tools for the SQF, QUES also has a capability to define a fully tailored version of the SQF. Thus, QUES can operate standalone and does not require ASQS to export a SQF definition.

QUES is capable of performing quality evaluations (scoring of questions and aggregating results up to the factor level according to the Guidebook Volume III procedures or any other hierarchical-based calculation procedure) at each of the four levels of DOD-STD-2167A system description (System, CSCI, CSC, CSU) for each of the nine DOD-STD-2167A life cycle phases. This storage and reporting facility supports the use of the SQM as a life-cycle monitoring method.

As an object-oriented database, QUES exhibits those capabilities and advantages associated with this class of advanced database systems.

QUES analytical capabilities should be upgraded with statistical models of software quality as they become available from the RL Software Quality Consortium.

5. SUMMARY

This section presents a summary of our findings organized around three major observations.

1. The Software Quality Methodology suffers from a lack of conceptual integrity.

On review, virtually every component of the SQM is open to criticism due to the presence of arbitrariness or subjectivity. The underlying SQM hierarchical model contains a second level layer which is outside of acquisition management concerns. Guidebook Volume II, the core of the SQM, is not fully specified and therefore open to undesirable interpretation and skepticism. Volume II also contains unsupported assertions concerning factor and criteria interrelationships and their respective magnitudes. The metric questions, the only direct points of measurement in the SQM, are a mixture of design checklists, development standards checklists and 'true' metrics (e.g., measures of program structure). Since the metric questions are the lowest hierarchical level in the model, their non-homogeneity implies that the higher levels of the model also reflect divergence from the goal of direct product measurement. The lack of a method for selecting metric questions, the equal weighting of metric questions in the scoring calculation, and the availability of criteria weighting cast doubt on what is being measured and to what extent the reported scores are meaningful.

The net effect of this lack of conceptual integrity is a reduced degree of confidence in the Software Quality Methodology and an impairment to acceptance and use.

Some of the issues noted here have been previously identified and analyzed in the validation studies. A compendium of issues and discussion is available in [17]. At a minimum, we recommend that more complete procedural guidance be placed into Guidebook Volume II. A good starting point is the analysis and detailed recommendations found in [16].

2. The Software Quality Framework is not evolving at a rate sufficient to adequately reflect the evolution of software engineering practice and DoD software strategy.

In order to be viewed as useful, the SQF must be continually revised in a timely manner to reflect actual contractor software development practice. In the contracting community, software development practice is strongly impacted not only by DoD mandates (Ada) but also by DoD stated areas of emphasis and strategy (reuse).

The SQF is lagging in this respect. Initial background work on a new SAFETY factor and on incorporation of object-oriented technology is included in sections III and IV of this report. Timely attention must be given to import reusability. It is likely that formal methods will increase in importance and use in the near term.

We recommend that a periodic formal review of the SQF be undertaken in order to identify new and emerging practices, mandates and directions which will require SQF revisions.

3. The specification of software quality should be based on formal mission area analysis.

A review of the initial effort in satellite mission area analysis demonstrates the value and necessity of this type of software quality analysis [22]. Although expensive, difficult and time-consuming, detailed software quality mission area analyses will substantially improve the usefulness and acceptance of SQM.

Such analyses will also provide the information required to generate mission area specific versions of the SQF. In its present form, the SQF is too abstract for use by individuals with operational concerns. Mission area specific versions should, at a minimum, use terminology specific to the mission area. An initial starting point would be to produce mission area specific guidebooks. Future development would concentrate on the creation of mission specific SQFs.

6. FUTURE RESEARCH

In this section, we sketch out a software quality research architecture which is suggested by the present study. The resultant capabilities are intended to define a second generation SQM (Software Quality Maximizer) which emphasizes quality specification and method-specific measurement. With the exception of modest modifications to QUES, each architectural component represents a substantial research effort. The architecture is principally composed of:

1. Deep software quality mission area analyses
2. Development of ESSQU (Experimental System for Software Quality)
3. Development of QUMAX (Quality Maximizer), a support tool for conducting technical and cost tradeoff analysis
4. QUCOST (Quality Cost), a support tool for determining cost of quality
5. Definition of QUSPECS (Quality Method Specifications) which encapsulate key quality factor parameters
6. QUES (Quality Evaluation System)

6.1 Architecture

This section sketches out how the software quality maximizer architecture is envisioned as a means of increasing the quality of delivered software products.

The mission area analyses are primarily needed to identify mission specific quality requirements and their respective ideal quality goal targets. Secondarily, these mission area studies can be used to develop mission specific SQFs.

ESSQU is an experimental Ada software testbed intended to help determine the real world interrelationships which exist among the software quality factors. These empirically defined interrelationships define the central logical driver of QUMAX.

The ideal quality goal targets from the mission area analyses are input to the QUCOST tool to determine an overall cost of quality and to allocate the overall cost to the quality factors by lifecycle phase.

The QUMAX tool then determines an internally consistent quality factor set. Iteration is likely at this point as sensitivity analyses are used to determine maximally achievable sets of quality levels under different funding level assumptions. The output is the initial quality specification (or sets of specifications for later use under different funding scenarios) for a specific mission area. (This process is carried out to the same level of system description [CSCI, CSC, CSU] as was developed in the mission area analyses.)

The QUSPECS relate factors to specific software engineering practices or methods known to have positive effects on software quality. The general form of a QUSPEC template (here oversimplified) is:

```
Quality Method Specification (FACTOR);

  goal level;
  life cycle phase [development];
  method,
    required resources,
      manpower (size),
      schedule (size),
    measures;

End QMS (FACTOR);
```

The QUSPEC concept is an attempt to relate concretely the achievement of quality to methods and their respective average costs and schedule requirements. A system size parameter is used to calibrate required resources to estimated system size.

At this point, the acquisition manager and the development contractor discuss the contractor's plans for software quality related tasks during the initial life-cycle phases. The plans are expressed in terms of QUSPECs. The QUSPECs and the QUCOST results are exported to QUES, which accesses the mission area quality specification and generates a tailored SQF for the lifecycle phase(s) whose quality plans are completed.

Note that in the Software Quality Maximizer methodology, measures are specific to factor/goal level/life cycle phase/method. This is done to increase the expected information value of the measurements. In addition, actual resource consumption is compared to expected resource consumption (estimated by) QUCOST. The former is analogous to metrics while the later is analogous to indicators.

As the contractor proceeds through the design phases of the lifecycle, more specific QUSPECs are used by QUES to generate additional phase dependent SQFs. Checkpoint measurement at both levels continues to take place.

6.2 Philosophy of Use

We have knowingly omitted commentary on whether or not this approach can be considered a software quality *measurement* methodology. The careful reader who is familiar with the RL SQM will see opportunities for using the approach described here in a quantitative manner. The final factor goals can be translated to numeric ranges, the method-specific measures can be treated in the same way as the SQM's metric/question structure, and scores can be developed and compared to goals. This philosophy has been termed the aggregate scoring model [21].

A different philosophy will place emphasis not on achieving acceptable scores but rather on executing the methods at high levels of software development practice. One can still use a question checklist but as method exit criteria rather than as inputs to a scoring algorithm. Something as straightforward as the requirement to document an instance of non-compliance with the exit criteria and then to require a go/no go decision by the customer's representative can be an effective driver of software quality improvement. This point of view is in sympathy with the requirements compliance[4] model [21].

7. LIST OF RECOMMENDATIONS

Recommendation 1. Perform an in depth analysis of the consequences of retaining versus deleting the criteria level. (section 2.3.1)

Recommendation 2. Based on prior evaluations, DoD directions, and current technology, modify the Guidebook definition of the basic SQF. (section 2.3.2)

Recommendation 3. Initiate an effort to build an Ada-based software quality testbed in order to experimentally determine factor and criteria interrelationships and their associated magnitudes. (section 2.3.3)

Recommendation 4A. Develop a rationale and procedure for the selection of metric element questions. (section 2.3.4)

Recommendation 4B. Determine the cost/benefit of creating, for each metric, one set of metric element questions which reflect the technical requirements necessary to achieve each possible factor level. (section 2.3.4)

Recommendation 5. Create the missing guidance sections and revise Volume II accordingly. (section 3.2)

Recommendation 6. Initiate additional mission area domain analyses. (section 3.4.1)

Recommendation 7A. Initiate an effort to develop a tradeoff assistance tool which would provide support for resolving factor/criteria interrelationships subject to cost constraints. (section 3.4.2)

Recommendation 7B. Initiate an effort to collect basic cost of quality data. (section 3.4.2)

Recommendation 7C. Initiate an effort to develop a cost of quality model.

Recommendation 8. Stop development work on the current ASQS concept. (section 4.1.1)

Recommendation 9. In the event that our recommendation 8 is not accepted, determine the scope of the ASQS knowledge base which would be required to generate meaningful mission area specific factor goal assignments. (section 4.1.1)

[4]It has been suggested that adherence is a more suitable term than compliance.

8. REFERENCES

[1] *DoD Critical Technologies Plan*, Report for the Congressional Committees on Armed Services, 15 March 1989.

[2] HR Subcommittee on Investigations and Oversight, *Bugs in the Program*, April, 1990.

[3] Jane Siegel, et. al., *National Software Capacity: Near-Term Study*, Software Engineering Institute, CMU/SEI-90-TR-12, May, 1990.

[4] Barry Boehm, J. R. Brown and M. Liplow, "Quantitative Evaluation of Software Quality"' *Proceedings of the 2nd International Conference on Software Engineering*, IEEE, September, 1976, pp. 592-605.

[5] Watts S. Humphrey, Terry R. Synder and Ronald R. Willis, "Software process Improvement at Hughes Aircraft", *IEEE Software*, (8,4), July, 1991, pp. 11-23.

[6] Richard DeMilo (interview), "Software Engineering Needs Visionaries, Multiple Approaches", *IEEE Software*, (8,1), January, 1991, pp. 92-93.

[7] Barbara Kitchenham, "Towards a constructive quality model Part I: Software quality modelling, measurement and prediction", *Software Engineering Journal*, July, 1987, pp. 105-113.

[8] Barbara Kitchenham and Lesley Pickard, "Towards a constructive quality model Part II: Statistical techniques for modelling software quality in the ESPRIT REQUEST project", *Software Engineering Journal*, July, 1987, pp. 105-126.

[9] Barbara A. Kitchenham and John G. Walker, "A quantitative approach to monitoring software development", *Software Engineering Journal*, January, 1989, pp. 2-13.

[10] S.G. Linkman, "Quantitative monitoring of software development by time-based and intercheckpoint monitoring", *Software Engineering Journal*, January, 1990, pp.43-49.

[11] Poul Grav Petersen, et. al., "Software Quality Drivers and Indicators", *Proceedings of the 22nd Hawaii International Conference on Systems Science*, IEEE, Jan. 3-6, 1989, pp. 210-218.

[12] IEEE Standard on Software Quality Measurement.

[13] T.P. Bowen , G. B. Wigle, and J. T. Tsai, "Specification of Software Quality Attributes", Volumes I, II, and III, RADC-TR-85-37, February, 1985.

[14] James L. Warthman. "Software Quality Measurement Demonstration Project I", RADC-TR-87-247, December, 1987.

[15] Patricia Pierce, Richard Hartley, and Sullen Worrells, "Software Quality Measurement Demonstration Project II", RADC-TR-87-164, October, 1987.

[16] Dynamics Research Corporation, "Software Quality Guidebook Validation Results", Contract No. F19628-84-D-0016, Task 25, CDRLS 104-107, Task 73, CDRLS 104-110, September 30, 1986.

[17] Scientific Applications International Corporation and Software Productivity Solutions, Inc., "Software Quality Framework Issues", TR (Interim) Volume II, 2 June 1989, prepared for Rome Air Development Center.

[18] Larry Kahn and Steve Keller, "The Assistant for Specifying the Quality of Software (ASQS) Operational Concept Document, RADC-TR-90-195 Vol I (of two), Sept, 1990.

[19] Larry Kahn and Steve Keller, "The Assistant for Specifying the Quality of Software (ASQS) User's Manual," RADC-TR-90-195 Vol II (of two), Sept, 1990.

[20] Software Productivity Solutions, Inc. "Software Requirements Specification for the Quality Evaluation System (QUES)," Contract No. F30602-88-C-0019, CDRL A002, September 5, 1989.

[21] Jeffrey A. Lasky, Alan R. Kaminsky, and Wade Boaz, "Software Quality Measurement Methodology Enhancements Study Results", RADC-TR-89-317, January 1990.

[22] Douglas Schaus, "Assistant for Specifying the Quality of Software (ASQS) Mission Area Analysis", RADC-TR-90-348, December 1990.

[23] Birsen Karpak and Stanley Zionts, *Proceedings of the NATO Advanced Institute on Multiple Criteria Decision Making and Risk Analysis Using Microcomputers*", NATO Advanced Science Institutes Series, Springer-Verlag, 1989.

SECTION III

SAFETY and SOFTWARE QUALITY

1. INTRODUCTION

Since 1976, Rome Laboratory has been funding the development of a software quality measurement methodology. During this time and up to the present, this research program has been the only sustained effort which has attempted to define a methodology to specify and evaluate software quality at the product level. The current definition of the methodology is found in a three volume set of guidebooks, *Specification of Software Quality Attributes* [14]. The key underlying concept of the methodology is a three level hierarchical model of software quality. The model is usually referenced as the Software Quality Framework (SQF). The top level is a set of thirteen customer-oriented *software quality factors*. The factor set is Efficiency, Integrity, Reliability, Survivability, Usability, Correctness, Maintainability, Verifiability, Expandability, Flexibility, Interoperability, Portability, and Reusability.

The second level is a larger set of defining attributes for the software quality factors. These are termed *software quality criteria* and reflect technical considerations of good software engineering and development practice. The third level is a still larger set of *software quality metrics* which are measures of the software quality criteria. The software quality metrics in turn are defined by lower level metric elements.

The methodology was defined at the time DoD-STD-2167 was in effect. The current standard is DoD-STD-2167A [13] which requires that a contractor perform a safety analysis "necessary to ensure that the software requirements, design, and operating procedures minimize the potential for hazardous conditions during the operational mission".

Rome Laboratory intends to periodically update the Software Quality Measurement methodology in order to be consistent with revisions to DoD-STD-2167 and to incorporate advances in software engineering technology. The aim of this study is to begin definition of a fourteenth software quality factor *SAFETY*.

The approach taken to define Safety is as follows. First, background reading was conducted in the area of software safety. Second, two important military standards, one each from the US and UK that are centrally concerned with software safety were reviewed in detail. Third, the factor, criteria definitions, and metric element questions from the software quality model were reviewed within the context of defining Safety. Fourth, based on the above, Safety was defined within the Software Quality Framework and a candidate set of metric element questions were created.

This paper is organized as follows. First, a brief discussion is presented which puts the requirement for a Safety quality factor in the light of current perspectives. This is followed by a discussion of the basic objectives of software safety and an outline of hazard analysis. Next, key military safety standards, MIL-STD-882B Notice 1 and MoD 00-55/1 are overviewed. This is followed by indepth discussions of both standards. It is hoped the background material will provide the reader sufficient background in order to better interpret and assess the final section's material on the Safety quality factor.

2. CONCERNS

System failures in real-time, safety-critical applications can result in death, serious injury, environmental damage or loss of property. Safety-critical systems are those where

> computers directly control the release or direction of energies capable of causing death or injury, and where the computer functions are so deeply embedded or must operate so rapidly that effective human oversight is not possible [1]

Examples of these applications are found in nuclear-based power generation, avionics and air traffic control, ground transportation control, and military weapon systems. The increasing use of software-intensive digital command and control systems in such environments is generating substantial worry and anxiety in the engineering, computer science and software engineering communities. The governments of the US, Canada and Western Europe have correspondingly increased their involvement in these areas by funding research and by including requirements for hazard analysis and safety verification in system development standards and regulations.

The general concern is that the scientific and technological base required to develop trustworthy[1] systems is not yet fully developed. The use of new technologies often increases operational risk. This risk increase becomes more pronounced when attempts are undertaken to use new technology (digital control systems in this case) and go beyond the accumulated experience base in some specific environment, safety-critical or otherwise. Even if digital control systems are being used only to replace more traditional mechanical or analogue control systems, the different and possibly unfamiliar failure modes of digital control systems may become problematical in regards to the design of fail-operational, fail-soft, or fail-safe mechanisms. One such different failure mode is a sudden complete failure (system hang-up), accompanied by unpredictable machine and data states.

Nonetheless, the lure of soft logic, with its advantages of change, information display flexibility and small, light weight, low power consumption platforms, is beginning to cause the proliferation of digital control systems in safety-critical environments. Many individuals familiar with the issues have concluded that the probability of catastrophic or serious accidents involving digital command and control of safety-critical systems is already too high[2,3].

In reaction to this state of affairs, the field of software safety engineering is beginning to emerge. Software safety engineering's foundations are based on classical safety engineering, especially the techniques collectively used to identify, categorize, prioritize

[1]David Parnas first used the term trustworthy when applied to high-integrity (which includes safety-critical) systems. He defines trustworthiness as 1- (probability of an undetected serious flaw remaining in the software after reviews and tests) [25].

[2]Perhaps the recent and massive telephone system failures in the US are an ominous warning.

[3]For the reader interested in following current developments in the safety-critical software field, the following are recommended: [a] the annual *COMPASS* (Computer Assurance) Conference held in the US; sponsored by IEEE, several government and military agencies, and industrial organizations. [b] the annual *SAFECOMP* conference held in the UK; organized by EWICS TC-7 (European Workshop on Industrial Control Systems). TC-7 is the Technical Committee on Reliability, Safety and Security. [c] *Risks to the Public*, a compendium of recently reported mishaps generally involving software problems, which regularly appears in Software Engineering Notes, the regular publication of the ACM Special Interest Group on Software Engineering. The contents of Risks to the Public are also available in the comp.risks newsgroup on USENET.

and analyze safety risks. However, due to the unique problems associated with software-intensive digital control systems, traditional safety analysis has been extended to include digital hardware and software. The next section generically describes this extension for software[4].

3. Software Safety

3.1 Definitions

Software safety engineering can be viewed as a process whose objective is to eliminate or reduce the risk that a software fault can cause a hazard by issuing erroneous (or not issuing correct) commands to the system it controls. This process includes the use of safety-oriented management practices, hazard analysis, determination of system redesign and/or safety feature requirements, and stringent V&V requirements. This process should be viewed both as an independent process which parallels the life-cycle model being used for development and as a process fully integrated into the life-cycle. The first view is important so that a continuous emphasis on safety is maintained throughout product development. The second view importantly recognizes that the safety objective is to create *safe systems* of which software is but one component and one consideration.

A *hazard* is a system state or set of conditions that, when combined with certain environmental conditions, is precedent to an accident. For example, consider a 4-way intersection controlled by a traffic light. If the light's digital control system fails for any reason and the traffic light's signals are all simultaneously green, then a hazardous condition exists. Note that this does not imply that an accident will definitely occur.

From a finite state machine point of view, a software-controlled safety-critical system can be in one of three states [27]:

1. a *safe* state where the outputs of the control system (e.g., commands to hardware devices, information displays) pose no threats.

2. a *hazardous* state where the control system's outputs pose a potential threat, and

3. an *unsafe* state where the control system's outputs cause harm.

Two principal objectives of safety programs then, including software safety programs, are (1) to prevent the system from reaching a hazardous state, and (2) to guarantee that if a hazardous state is reached that there is always a state transition from the hazardous state back to a safe state.

A *safety feature* is a system design requirement whose purpose is to eliminate unacceptable safety risks and constrain other less severe risks to an acceptable level. In the traffic light example, an appropriate safety feature would be some type of interlock process which would logically or physically prevent an all green condition. If the mechanism which supports the interlock process fails, the traffic light would then blink red in all directions. That is, the system would fail into a safe state. The design of a software-based or software-controlled interlock mechanism would have to (a) include the possibility of an

[4]Substantial activity has begun in computer hardware safety analysis as well. For example, see [26] for recent findings and developments from the VIPER project (Verifiable Integrated Processor for Enhanced Reliability)

interlock failure, (b) include code to detect interlock failure, and (c) include code to change all lights to blink red. The (b) and (c) categories of software, together with software controlling the light, are *safety-critical* since their failure would create a hazardous condition.

The next section presents some basic material describing hazard analysis, which is the starting activity for virtually all system safety programs.

3.2 Hazard Analysis

Hazard analysis is a set of activities which (a) systematically examines systems to identify potential hazards, their causes, severities and probabilities of occurrence and (b) establishes requirements for safety features to eliminate or control these hazards.

When a real-time, embedded control system failure occurs, the process or event under software control typically cannot be simply terminated. Instead, safety-critical embedded control systems may be designed to:

(a) be *fail-operational* (fault-tolerant) and continue to provide full performance and operational capabilities, or

(b) be *fail-soft* and continue to operate but provide either reduced performance, reduced capabilities or both, or

(c) be *fail-safe* and limit the amount of damage caused by a failure and provide little or no operational capability except where necessary to ensure safety.

A hazard analysis is one important input into the overall design decision-making process. The result of a hazard analysis is actually a safety requirements specification, which will be subject to safety-related V&V analysis and tests. The safety feature requirements become additional system requirements, some of which are allocated to the software requirements specification.

Safety-critical system functions or system components implement safety features. Thus, the failure of a safety-critical function could give rise to an accident. *Safety-critical software* is software which implements or controls safety-critical functions.

Digital control systems can achieve hazardous states because of:

- human error,
- software control error,
- interface error,
- hardware failure, and
- environmental stress

Exhaustive hazard analysis is generally not possible, due to the combinatorics of the above hazard sources, and the realities of incomplete knowledge and limited investigative and design resources.

A hazard analysis is the beginning point for system safety considerations. Typically, hazard analysis follows a well-established mutli-stage approach. As noted previously, each of the following generic major hazard activities takes place within the context of the overall system development life-cycle phase.

1. Preliminary Hazard Identification.

As early as possible, optimally during the concept definition phase, a preliminary list of system hazards is created. This activity usually makes use of existing safety-related data available from the documentation and operational experiences of other similar systems, as well as any available system requirements analyses and safety standards and regulations.

2. Preliminary Hazard Analysis.

Based on the preliminary hazard identification, an evaluation of the major systems hazards is conducted and safety critical areas are identified. An objective of this phase is to begin to identify and begin to define system re-design requirements or safety-feature requirements[5].

Once the hazards have been identified, the link to software safety analysis and the resultant software safety requirements is generally provided by Software Fault Tree Analysis (SFTA) [15], Timed Petri Net Analysis (TPNA), [16], Safety Analysis Linkage Technique, (SALT) [17], or other available modeling and analysis techniques. The hazards subject to these types of analyses are the hazards resulting directly from software execution or non-execution, solely or in combination with other factors.

These initial high-level assessments are made within frameworks such as shown in Tables I and II. As in any other project activity, resources for hazard analysis are limited. The classification of the identified high level hazards directs resources to areas of greatest concern. The definitions shown are not intended to be absolutes, but rather are starting points for tailoring the hazard analysis to the system requirements and operating environments of interest. Often, the definitions of the probability ranges shown in Table II are given quantitatively and in frequencies which are meaningful to the task at hand. A typical approach is that the mid-points of adjacent probability ranges are separated by one or two orders of magnitude, with the possible exception of the probability of *incredible*. The assignment of extremely low occurrence probabilities (e.g., 10^{-9}) may not be meaningful. In such cases, a qualitative definition may be more useful. A general trend in the field cautions against over-reliance on the absolute values of the assigned probabilities[6].

Most system safety standards categorize hazard severity and probability in similar ways. As an example, Tables I and II contain definitions taken from the two key US and

[5]In safety engineering, there is a well-established system safety precedence: (a) design to eliminate risk, (b) incorporate safety devices if identified hazards cannot be eliminated, (c) provide warning if (a) and (b) are not feasible, and lastly (d) develop procedures and training. As an example of (a) consider the traffic light example. The risk (collision) that the safety device (traffic light) substantially reduces could also be eliminated by (re)designing all roads to be parallel. As an example of (c), consider a tidal wave. In order to satisfy (a), we would have to constrain habitants to areas that have no boundaries with the oceans. The best we can probably do here is to provide fast transport to high ground (b) together with an early warning system (c).

[6]A regrettably example of over-reliance was provided by R.P. Feynman, a scientist contributor to the minority opinion included in the report of the Challenger tragedy. "It appears that there are enormous differences of opinion as to the probability of a failure with loss of vehicle and of human life. The estimates range from roughly 1 in 100 to 1 in 100,000. The higher figures come from the working engineers, and the very low figures from management. What are the causes and consequences of this lack of agreement. Since 1 part in 100,000 would imply that one could put a Shuttle up each day for 300 years expecting to lose only one, we could properly ask "What is the cause of management's fantastic faith in the machinery?".

UK military standards discussed in the next section. (The definitions are essentially identical in both standards)

TABLE I -- Hazard (Accident) Severity

Description	Definition
CATASTROPHIC	Multiple deaths or system loss
CRITICAL	A single death; and/or multiple severe injuries or severe occupational illness
MARGINAL	A single severe injury or occupational illness; and/or multiple minor injuries
NEGLIGIBLE	At most a single minor injury

TABLE II -- Hazard (Accident) Probability (during operational life)

Description	Definition
FREQUENT	Likely to occur frequently
PROBABLE	Likely to occur often
OCCASIONAL	Likely to occur several times
REMOTE	Likely to occur some time
IMPROBABLE	Unlikely, but may exceptionally occur
INCREDIBLE	Extremely unlikely that event will occur, given assumptions about domain/system

3. System/Subsystem Hazard Analysis

The system hazard analysis determines how system operations and failure modes can affect the safety of the total system (and subsystems if present). A review is conducted to assess compliance with safety-related items in the system/subsystem requirements documents.

The system level study may be preceded by a subsystem and interface hazard analysis. Functional and component failure analysis are representative techniques used for system hazard analysis. *Functional analysis* identifies hazards, at the system and component level which may result from (a) normal performance, (b) specified degraded performance, (c) incorrect functioning and (d) absence of function. *Component failure analysis* identifies single failures that contribute to known hazards and single failures that create new, additional hazards.

4. System Risk Analysis

Hazard risk is the product of hazard severity and hazard probability. The concept of hazard risk helps to prioritize the allocation of resources in order to maximize system safety by providing guidance into the design process.

Hazard severity and probability from Tables I and II are combined in Table III to assign risk classes to all combinations of the two factors. The risk classes are then defined in Table IV in terms of acceptability of the risk. For example, since class A risks are unacceptable, a determination that an identified hazard is Class A would mandate that safety features, of sufficient design to remove the hazard from Class A, become verifiable system requirements.

TABLE III -- Risk Classification of Hazards (Accidents)

	Catastrophic	Critical	Marginal	Negligible
Frequent	A	A	A	B
Probable	A	A	B	C
Occasional	A	B	C	C
Remote	B	C	C	D
Improbable	C	C	D	D
Incredible	D	D	D	D

TABLE IV -- Risk Classification Interpretation

Risk Class	Interpretation
Class A	Intolerable/Unacceptable
Class B	Undesirable, and acceptable only by decision of the safety review authority.
Class C	Tolerable/Acceptable with concurrence of safety review authority
Class D	Tolerable/Acceptable with concurrence of normal project review

4. Safety-Critical Software Development Standards (Military)

As noted previously, it has become apparent, both within and outside of the military, that there is a need for major improvements in software engineering practice as applied to digital controlled safety-critical applications. The remainder of this paper focuses on the major US and UK safety standards that set development requirements for software to be used in safety-critical systems. This section suggests that there has been a noticeable progression during the past fifteen years of requirements for the development of safety-critical military systems as evidenced by the scope, depth and technical methods specified in military development standards. This progression parallels the rapidly increasing amounts of software used in military weapons systems.

4.1 First generation

In the military arena, the clear and irreversible trend is toward software-intensive military systems. For some time, several militaries have promulgated design and development standards which specifically address system safety, including systems containing digital computers. (Note that these older standards were issued in times when the amount of software used in weapons systems was far less compared to the present). Many of these standards are based on long-standing industrial safety practice and include requirements for safety program definition and management, engineering management and specific technical analysis.

Some standards specifically mention requirements for software safety analysis, but these are often stated in broad and general terms. A representative example from the US military is MIL-STD-1574A (USAF), *System Safety Program for Space and Missile Systems,* 15 August 1979. This standard is a tailored application of MIL-STD-882A (see below) for space, missile and related systems. There are two software requirements, both almost identical in their wording. One is the requirement to conduct a Software Safety Analysis. the other to conduct an Integrated Software Safety Analysis. The software subject to this standard would include, for example, both flight and ground control systems as well as diagnostics.

4.2 Second generation

The major US systems and software safety standard[7] is MIL-STD-882B Notice 1, *System Safety Program Requirements*, 1 July 1987. This is an upgraded standard which originally focused on hardware safety analysis but which now also contains detailed software safety analysis requirements. Prior versions of MIL-STD-882 specified safety analysis requirements for Program Management and Control ("100 series tasks") and Design and Evaluation ("200 series tasks"). The B Revision Notice 1 integrated some new software analysis safety requirements into the hardware-oriented 200 series tasks, and added a 300 series set of tasks specifying the required types of software safety analyses[8].

[7]There are also specific safety standards related to software development for nuclear weapons systems. See, for example, AFR 122-9, The Nuclear Safety Cross-Check Analysis and Certification Program for Weapon Systems Software, and MIL-STD-SNS (Navy), Software Nuclear Safety.

[8]Work is underway on the C revision to 882B. The planned major structural change is the integration of the 300 series tasks into what are now identified as the 200 series system-level tasks. The 300 series will then disappear as a separate entity. The intent of this change is to preserve the standard's initial system safety perspective, whose overall cohesiveness was somewhat impaired by the addition of the 300 task series. Other central changes include (1) placing greater emphasis on system safety testing, and (2)

The standard also includes a substantial amount of interpretation, implementation and technical guidance for the three categories of tasks. The "300 series" tasks were designed to be integrated into the software development process defined by DoD-STD-2167A. Table V shows the standard's task structure. The standard encourages the tailoring (modifying) of task requirements to meet specific conditions and circumstances.

introducing a new detailed critical function analysis activity which will focus, in part, on the rigorous analysis and validation of time-critical execution sequences and dependencies.

TABLE V Task Structure of MIL-STD-882B Notice 1.

Series 100 Tasks

100	System Safety Program
101	System Safety Program Plan
102	Integration/Management of Subcontractors
103	System Safety Program Reviews
104	System Safety Group
105	Hazard Tracking and Risk Resolution
106	Test and Evaluation Safety
107	System Safety Progress Summary
108	Qualifications of Key System Safety Engineers/Managers

Series 200 Tasks

201	Preliminary Hazard List
202	Preliminary Hazard Analysis
203	Subsystem Hazard Analysis
204	System Hazard Analysis
205	Operating and Support Hazard Analysis
206	Occupational Health Hazard Assessment
207	Safety Verification
208	Training
209	Safety Assessment
210	Safety Compliance Assessment
211	Safety Review of Engineering Change Proposals and Requests for Deviation/Waiver

Series 300 Tasks

301	Software Requirements Hazards Analysis
302	Top-Level Design Hazards Analysis
303	Detailed Design Hazards Analysis
304	Code Level Hazards Analysis
305	Software Safety Testing
306	Software/User Interface Analysis
307	Software Change Hazards Analysis

4.3 Third Generation

An accelerating movement within the software development research and technical community is to rely substantially on mathematical-based formal methods to provide a more rigorous basis for the specification, design and validation of software systems. Over the past decade, much progress has been made in developing usable versions of formal methods technology, including in some cases, accompanying toolkits. Formal methods technology is not currently in widespread use and substantial research and development issues remain to be resolved. Nevertheless, the UK Ministry of Defence's (MoD) decision to base their new safety-critical software development standard on formal methods technology has gathered very substantial attention from the North American and Western European technical communities. Although the initial draft of the standard was first circulated to a relatively small audience for comment, several thousand copies had been requested by the end of 1989.

These new standards are regarded by many as a breakthrough or landmark in safety-critical software development practice, due primarily to the standard's reliance on and mandated use of formal methods[9]. From the MoD's point of view, the technical problems of system verification and validation were becoming progressively more difficult [1]. The seemingly endless price/performance gains in computer hardware were now permitting more complex and numerous functions to be integrated and be implemented economically under software control. This increase in the level of system and software design complexity was viewed as increasing the probability that a subtle but potentially catastrophic design error would remain implemented in a deployed safety-critical system. Traditional specification and design techniques were judged as having inadequate resolution power in the military safety-critical environment. Consequently, the decision was made to employ the higher resolution power of mathematics-based formal methods[10]. It is important to note that the standard also relies on more traditional software engineering technologies such as structured techniques, static path analysis and dynamic testing.

The first standard, MoD 00-55/1, *The Procurement of Safety Critical Software in Defence Equipment, Part I Requirements*, 5 April 1991, is the standard receiving most of the attention. There is also a *Part II Guidance* (same title and issue date) which provides interpretation, implementation and technical consultation for selected requirements of Part I. Also issued simultaneously is MoD 00-56/1, *Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment*, 5 April 1991. (It is interesting to note that the 00-56 standard makes a reference only once to *software*.) Both standards were first issued in 1988 as Draft Interim Standards for exposure and comment. The most recent issues cited above are classified as Interim Standards.

[9]This is not the first time that a government systems standard has required the use of formal methods. The DoD's Trusted Computer System Evaluation Criteria, popularly known as the *Orange Book*, requires formal proofs for A1 level systems that the specifications are consistent with the requirements [29].

[10]Another reported reason for the adoption of formal methods was the virtually demise of the traditional watchmaking industry. That industry had provided the precision mechanisms for the arming and fusing of explosives. In the UK, the Ordnance Board, who must pass judgement on the integrity of ordnance explosive control, would not endorse the application of traditional computer software development practices for the purpose of implementing software-controlled arming mechanisms.[1]

4.4 Analyzability

The design and implementation of safety-critical software will generally be carried out at a higher standard of practice than the development of non safety-critical software. The introduction of formal methods into a safety standard raised an important question concerning the determination as to what is considered acceptable software engineering practice for safety-critical software. MIL-STD-882B Notice 1 states, (in Appendix A, Guidance, p. 19), that "the application of good software engineering practices is vital to designing software which is safe and analyzable. No additional explanation, criteria or tests are suggested to determine whether a practice is or is not analyzable. Absent specific definition or guidance, the most likely operational interpretation would be: is the practice readily amenable to analysis.

In MoD 00-55/1, a standard of analyzability was eventually used to resolve the most controversial part of the standard found in the prior draft versions, namely the list of Unacceptable and Prohibited Practices. These system design and programming practices were absolutely prohibited. The reason for the prohibitions was not that the practices were necessarily intrinsically dangerous (although the draft standard did use the term unsafe), but rather that their use at this stage of formal development technology would impair or preclude rigorous software analysis and validation. Included in this list were:

- floating-point arithmetic
- recursion
- interrupts, except for a fixed-interval timer
- assembly language
- dependence on parallel asynchronous processing
- multi-tasking
- object code patching
- dynamic program reconfiguration
- dynamic memory management

As might be expected, the banned practices generated considerable comment from the contracting community which had been routinely using the practices for decades. In order to clarify the issue, the MoD introduced a strict standard of analyzability and an accompanying test of compliance in the latest version of the standard. The test is found within the more detailed discussion of the standard in section 6.1 below.

In certain cases, if the strict standard cannot be met, a somewhat weaker standard of assurance may be available. This in effect is a relaxation of the strict form of analyzability, although a close reading of the standard suggests that these practices would still be considered problematical, both for design and implementation but also for performance analysis[11]. One effect of the analyzability standard is to render the list of prohibited practices unnecessary; indeed, the list is absent from the current issue. (In the specific case of assembly language, its use is now permitted with a recommendation for minimal use, not to exceed twenty to fifty lines per insert). As formal methods technology develops,

[11]The concepts of analyzability and assurance, outside of proof requirements, also reach into considerations of performance. MoD 00-55 (Requirements)/1 states in section 30.1.3, page 16, "SCS (Safety Critical Software) shall be designed so that it is easy to justify that it meets its specifications in terms of both functionality and performance. This requirement may restrict the length and complexity of the software and inhibit the use of concurrency, interrupts, floating point arithmetic, recursion, partitioning and memory management. MoD 00-55 (Guidance)/1 continues, in 30.5.1.2, "The straightforward architectures produced according to the requirements for analyzability ... greatly *simplify performance analysis*.

additional practices once found on the prohibited list will likely become available for development use.

In sum, the intent of the analyzability standard is to facilitate and simplify verification and validation.

4.5 Predictability

An overriding concern of both standards is that the behavior of the software-controlled system be predictable under all circumstances. This ultimately permits judgements to be made about the safety of the system and provides the basis for the system's safety certification and/or acceptance.

At the code level, predictability is enhanced by proving, or more weakly demonstrating, that the compiled object code implements the semantics (meaning) of the source code. There are at least three sources of unpredictability at the program level; that is circumstances where the object code does not implement the source code semantics.

1. Use of an optimizing compiler. Optimizing compilers substitute or rearrange instructions generated by an earlier compilation pass. Whether the goal of the optimization is to increase execution speed or reduce memory requirements, the optimization process can destroy the absolute equivalence between source and object code. Thus, in safety-critical applications, the use of an optimizing compiler is in direct conflict with the safety principle of predictability.

2. Unintended side effects of language features and programming practices. The logical consequences stemming from the use of certain language features can also introduce unpredictability into the execution stream. A typical example occurs in the evaluation of boolean expressions, where the expression on the right hand side of the logical operator is a function, which when evaluated can either raise an exception condition or inadvertently change the value of a global variable (the later could occur if a function's argument has been declared **in out**. The execution behavior of the system may be different depending on whether the logical operator is an **and** (**or**) in one case and an **and then** (**or else**) in another case.

3. Undefined language feature implementation. An example here is Ada tasking, although the issue is the same for most concurrency-related features. Since the compiler designer is free to use either time-sliced round robin or highest priority first scheduling, and keeping in mind the many variants of each, unpredictability could arise just due to the many different compiler implementation choices.

In sum, the danger which may result from these code level sources of unpredictability argues strongly for the use of restricted language subsets with well-defined, formal semantics for the development of safety-critical software. Here, we again see the importance of the analyzability principle, since a program written in a language with formal semantics is, at least in theory amenable to formal analysis as well as to static analysis. The later type of analysis, performed at the source level by another program, potentially yields predictability-related information such as undesirable source code attributes such as unreachable code, uninitialized variables, and unintended dependencies between input variables and output variables.

4.6 Comparison of US and UK Standards

The two standards are in a sense equivalent in that they both attempt to mandate practices and procedures which have the objective of reducing the probability that the occurrence of an accident in a safety-critical environment is due to a software deficiency or failure. Both standards focus centrally on safety-critical software components and basically ignore non-safety critical software. They both approach hazard analysis in roughly the same way and each has extensive reporting and safety review requirements[12].

There are naturally differences as well. The US standard stresses an overall integrated systems level approach while the UK standard focuses almost exclusively on software. The UK standard is somewhat more formal in its administrative and organizational requirements. In general, more specific and mandated software-related technical requirements are to be found in the UK standard.

Of course, the major difference is MoD 00-55's mandated use of formal methods and formal proofs. Only time will tell what the magnitude and consequences of the formal methods approach will be in the safety-critical domain, but the issuance of the standard itself is clearly significant. Nothing in the structure or objectives of DoD 882B precludes the future incorporation of formal methods requirements.

5. MIL-STD-882B Notice 1--System Safety Program Requirements

5.1 Overview

Series 100 tasks establish the requirements for a system safety program and set forth required organizational structures and reporting requirements. Series 200 tasks are primarily focused on hazard analyses.

For safety concerns related to software, the standard identifies and defines a subset of all the system's software which will receive special attention throughout the system development cycle. That is, the 300 series tasks apply only to this special subset termed *Safety Critical Computer Software Components (SCCSCs):*

> are those computer software components (processes, functions, values or computer program states) whose errors (inadvertent or unauthorized occurrence, failure to occur when required, occurrence out of sequence, occurrence in combination with other functions, or erroneous value) can result in a potential hazard, or loss of predictability or loss of control of the system [3].

Typically, SCCSCs interface with safety critical functions in hardware, such as motion controls or display of safety critical data. Thus, the basic process for initially identifying SCCSCs is to trace from the safety critical hardware functions to the software *execution path* which ultimately controls the critical hardware functions[13]. All software on this execution path whose failure could cause the system to behave in a dangerous or

[12]These and other similarities are not surprising since there is close informal collaboration on system and software safety standards between the US and the UK.

[13]"Normally, only software that exercises direct command and control over the condition or state of the hardware components or can monitor the state of the hardware components are considered critical from a safety viewpoint", *USAF Software Safety Handbook*, [18].

harmful way is considered to be SCCSC. In addition, all functions involved in the creation of SCCSCs (e.g., creating a table and/or table values) or that use the SCCSCs are also considered to be SCCSCs.

The standard does not apply to non-SCCSC software. This exclusionary focus on SCCSCs suggests the possibility that non-SCCSCs could adversely impact the safety critical software. MoD 00-55 adopts the same perspective. We discuss this important issue in section 7 following a description of the UK standards.

5.2 Series 300 tasks

This section describes each of the series 300 software tasks in some detail[14]. Since we are concerned here primarily with software issues, no descriptions of the series 100 or 200 tasks are included.

The 300 task series is designed to be integrated with the software development cycle specified by DoD-STD-2167A. In effect, each of the eight major activities which define the DoD-STD-2167A development process is expanded to include safety analysis This is a potentially effective and efficient approach, since the safety analysis is not considered as a separate, standalone activity.

Task 301 - Software Requirements Hazard Analysis (SRHA).

The purpose of the SRHA is to develop safety design requirements to be included in the system and software preliminary design. Input into Task 301 is the Preliminary Hazards List (Task 201), the Preliminary Hazards Analysis (Task 202), the SSS, the SSDD, the SRS, the IRS and other previously developed policies and guidelines applicable to the system. The SRHA takes place during System Requirements Analysis/Design and may overlap with the Software Requirements Analysis. The standard requires that a software safety requirements tracking system (e.g., a Safety-Critical Requirements Traceability Matrix) be established within the configuration management system. An important outcome of the SRHA is to provide assurance that all system safety requirements have been translated into software requirements and documented in the SRS.

Task 302 - Top Level (Preliminary) Design Hazard Analysis (TLDHA)[15].

The TLDHA takes place during Preliminary Design. The purpose of the TLDHA is to allocate the identified safety design requirements from the SRHA into the SDD (Software Design Document) for each CSCI. Thus, specific hazards are allocated to specific CSCs which become designated as SCCSCs. One important outcome of the TLDHA is that the software is preliminarily partitioned into SCCSCs and non-SCCSCs. Analysis is also undertaken during this time to identify other potential SCCSC software; i.e., software which directly (e.g., through an invocation) or indirectly (e.g., via a table which a SCCSC accesses or creates) influences a SCCSC. All SCCSCs identified during TLDHA are placed under configuration control (entered into the Safety-Critical Requirements Traceability Matrix).

[14]This section closely follows the analysis found in Michael L. Brown, "Software Systems Safety and Human Errors," *Compass '88 Proceedings of the 3rd Annual Conference on Computer Assurance*, IEEE, 1988.
[15]The term "top-level" reflects DoD-STD-2167 terminology and will likely be replaced in 882C.

The SCCSCs are prioritized by a safety hazard risk analysis (see Tables III and IV). Typically, at least all class A and class B (excluding class B Negligible/Frequent) risks continue to be subject to the full requirements of the standard, although the specifics will depend on the tailored standard. Preliminary safety test requirements are developed for the Software Test Plan.

Task 303 - Detailed Design Hazards Analysis (DDHA).

The DDHA takes place during Detailed Design. The DDHA continues the refinement of the safety design requirements developed from the SHRA and the TLDHA. As in the TLDHA, additional SCCSCs may be identified. All SCCSCs identified during TLDHA are placed under configuration control (entered into the Requirements Traceability Matrix). It is important to note the Detailed Design phase culminates with the Critical Design Review at which time the software design is supposedly frozen. Thus, any safety design requirements not included prior to CDR will not likely be implemented. Special attention, then, should be paid to reviewing the Requirements Traceability Matrix at this time. Specific safety test requirements, plans and procedures are also developed, as well as safety-related documentation to be included in the user, operator and diagnostic manuals.

Task 304 - Code Level Hazards Analysis (CLHA).

Page A-22 of the standard specifies the requirements for CLHA. One can view these requirements as micro, but comprehensive V&V activities applied to the individual SCCSCs. A variety of inspections, reviews and walkthroughs appear to be required by the standard, as well as process flow (static path) analysis and other V&V code level assurance techniques. Specific tests should be developed to verify the correct execution of any material safety features discrepancies identified during the CLHA. This is undoubtedly an expensive and time-consuming task for which no specific resource level guidance is provided by the standard.

Task 305 - Software Safety Testing (SST).

Software safety testing begins at Coding and CSU Testing and continues through CSC Integration and Testing, CSCI Testing and System Integration and Testing. The overall objective of SST is to provide assurance that the system will behave in a predictable and safe manner under all circumstances. In addition to the use of good testing practice, additional attention will be placed on testing safety critical functions and the SCCSCs. The test program should incorporate both naturally occurring and artificial boundary conditions which have repeatedly caused problems in the past. For example, the impact of the *sea level* condition on avionics systems, or more generally, unanticipated zero and negative program variable values. In addition, all credible failure modes must be tested. Many of these tests would fall within the existing 2167A stress testing requirement (general requirement 4.3). The overall objective of SST is to provide assurance that implemented safety requirements accomplish the intent of the specified requirements.

Task 306 - Software/User Interface Analysis (SUIA)

Not all safety hazards may have been eliminated or deemed controllable by the design/implementation. The user/operator interface is reviewed to ensure that the capabilities provided by the interface support safe and complementary system operation. Specifically, the standard requires design recommendations which provide for (a) unambiguous and complete display of safety critical information, (b) the detection of hazardous conditions, (c) operator warnings to alert the user to abnormal conditions or errors, safe cancellation of a process or event, and (e) safe survival and recovery fro a detected hazardous condition.

Brown [8] strongly recommends that human factors analysts get involved as early as possible during the definition and design of safety-critical systems.

Task 307 - Software Change Hazard Analysis (SCHA)

Changes, either during development or post-deployment, may cause an otherwise safe system to become unsafe. Development changes may be generated from any baselined software development product or software deficiency reports. Post-deployment changes should strictly include all software maintenance and software adaptation (evolution) activities.

The standard requires a strict and comprehensive change analysis which is similar in principle to regression testing. The (re)application of hazard analysis and testing begins at the highest specification level that the change effects and proceeds forward into standard's task sequence. For example, if a change occurs at the system level, the entire sequence of analyses, Tasks 301 through 306 must be performed. Special attention is paid to the impact of change on existing SCCSCs and the possibility that new SCCSCs are created in the process of integrating the change into the system. In the later case, all of the standards requirements specific to SCCSCs will have to be met.

6. MoD 00-55/1

6.1 Overview

MoD 00-55 Part 1 and Part 2 set forth, in considerable detail, the software engineering requirements for safety-critical software (SCS)[16]. All software is considered to be safety-critical unless shown otherwise by the companion hazard analysis and risk classification standard MoD 00-56.

The standard requires the use of formal methods for specification, design and verification in place of, to the greatest extent possible, more traditional and less rigorous methods. At the present time, several formal methods, most notably VDM [19] and Z [20] are sufficiently complete, well-developed and understood that they are usable by the development community. This is not true of all existing formal methods. There is also a clear need for substantial, long-term formal methods research and development activities.

The general approach employed by the standard is to require formal verifications at each stage of the life-cycle. Each formal activity along the life-cycle chain that transforms the initial English statement of requirements into executable code is accompanied by *proof obligations*. These are formally stated logical assertions about the properties of the development object currently being created. Typically, development occurs in a series of successive refinements. Each refinement generates proof obligations which must be *discharged* (proved) to demonstrate correctness.

Discharge of a proof obligation is accomplished by *formal argument* which may be a complete formal proof or a partial proof known as a rigorous argument. In any event, from an assurance (and feasibility) perspective, it is desirable that tools be used to generate the

[16]The standard also contains requirements for safety program management.

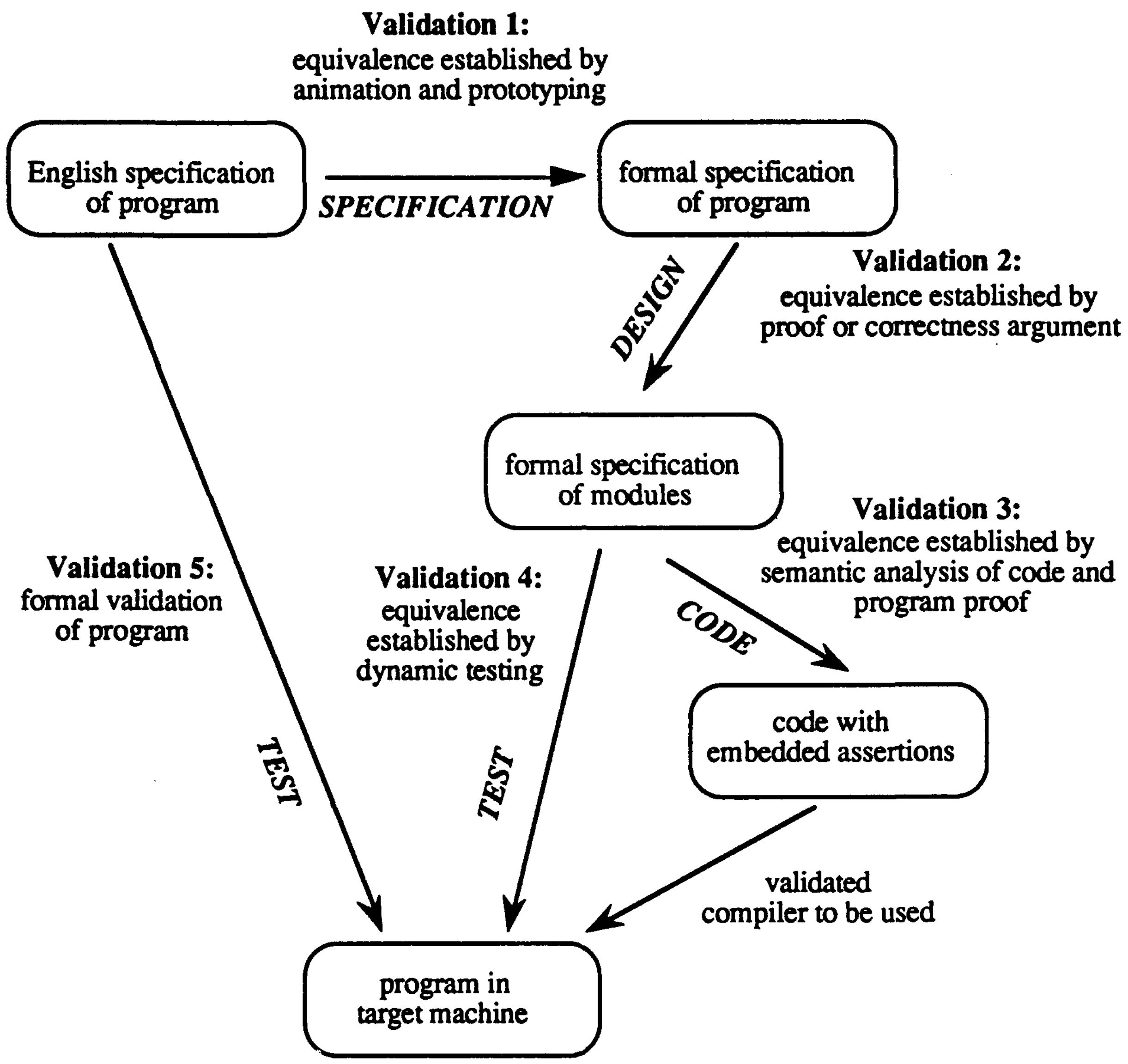
proof obligations and formal arguments. This becomes more desirable and eventually necessary as the size of the program increases. One such tool is a proof obligation generator. This tool embodies the proof theory for the method and will produce the proof obligations automatically when provided with the formal design. At the present time, a few proof obligation generators are commercially available, most of which are more available for formally-based programming rather than formally-based design. Attractive complementary tools to assist with discharging the proof obligations are theorem provers (typically interactive assistants) and proof checkers (programs which validate proofs). Again, only a few such tools are currently available.

The ability to demonstrate proof obligation discharge is the strict test of analyzability (see section 4.4 above). If a proof obligation for one of the undesirable practices can be generated and discharged, the practice will presumably be acceptable. Otherwise, it will remain prohibited unless the less strict test of assurance is acceptable to the safety review authority.

We see then, that the formal methods requirements are, in toto, beyond the state-of-the-practice, and in many cases beyond the state-of-the-art. The standard's requirements were purposely set beyond the current state-of-the-practice/art in order to stimulate research and commercial activity. Consequently, the MoD does not expect full compliance with the standard at this time. The language used in Part 2 (Guidance) reflects this expectation. As the state-of-the-art advances, more stringent compliance will be required.

6.2 Software Development

This section describes in some detail the organization and use of the standard's required software engineering practices for the development of safety-critical software (SCS). Figure 1 shows the triangle of reasoning for establishing program correctness. The overall 00-55 strategy for developing SCS is one of continual validation by the development team. Each validation is reviewed by a V&V team and then a Review Committee.



Validation 1:
equivalence established by animation and prototyping
English specification of program
SPECIFICATION
formal specification of program
DESIGN
Validation 2:
equivalence established by proof or correctness argument
formal specification of modules
Validation 3:
equivalence established by semantic analysis of code and program proof
CODE
code with embedded assertions
Validation 5:
formal validation of program
Validation 4:
equivalence established by dynamic testing
TEST
TEST
validated compiler to be used
program in target machine

Figure 1. Triangle of reasoning for establishing program correctness. Adapted from [9].

1. Establish an equivalence (animation) between the Software Requirements Specification expressed in English and the derived Software Specification expressed in a formal method.

2. Establish an equivalence (formal arguments) between the formal Software Specification and the derived Module Specifications expressed in a formal method.

3. Establish an equivalence (formal arguments, semantic analysis of source code) between the Module Specifications and the Source Code).

4. Establish an equivalence (dynamic testing) between the Module Specifications and the compiled Source Code.

5. Formally (officially) validate the software (testing) against the original Software Requirements Specification.

The remaining parts of this section provide more detailed descriptions of each of the five validations.

Validation 1.

The standard defines *animation* as the process by which the behavior defined by a formal specification is examined and validated against the informal (English) requirements. Animation is concerned with exploring the properties of the formal specification which is a mathematical model of the behavior of the SCS. The purpose of this first validation is requirements capture, not verification of the translation process.

Animation is carried out by use of both formal arguments and an executable prototype. The objectives of the animation are:

(a) to assess the completeness and internal consistency of the formal specification,

(b) to identify incompletely defined functionality or the omission of functionality,

(c) to identify errors in the specification that lead to failure conditions,

(d) to establish the behavioral boundaries of the formal specification, and

(e) to explore and assess extensions and improvements.

Formal arguments are used to show that the safety features specified in the Software Requirements Specification are present in the formal specification. The executable prototype, optimally derived automatically from the formal specification, is used to check the behavior of the specification, testing as a minimum of safety features of the SCS. Other means of producing the executable specification include translation into a logic programming language or into a language which supports the abstract data types used in the formal specification. There is no expectation that the executable prototype will be complete in all respects. Formal arguments are required to show the equivalence of the executable prototype to the formal specification. The major difference in this validation between the 00-55 and more traditional standards is the use of formal arguments, as the use of executable prototypes is established technology.

Validation 2.

The requirements for the design process are very comprehensive and include a number of performance investigations that are outside the formal methods framework. The design itself, however, is created using a formal method together with proof obligations as described earlier. The standard emphasizes that the design should facilitate verifiability of both its functionality and performance requirements. It is here that certain of the previously prohibited practices may conflict with the emphasis on verifiability via analyzability. Also, the standard suggests an upper limit of 5,000 source lines to implement an individual design. An additional design requirement is that wherever possible, SCS should execute on a processor physical separate from the processor(s) used to execute non-SCS. In cases where this is not possible (space, weight and power constraints), software mechanisms can be used to isolate the two categories of software on a single processor. If these mechanisms are assessed to be safety-critical, they would be developed according to the requirements of the standard.

Defensive programming is required to be incorporated into the design. The standard takes the position that inputs from a formally proved module do not require checking but that all other inputs should be checked.

Performance analysis is to be carried at the point in the design process where sufficient detail is available to support high-level performance. Here also the requirement for analyzability is raised, this time from the point of view that the use of simple, straightforward architectures, with deterministic scheduling and little or no concurrency will greatly simplify performance analysis. The standard suggests that hardware resource utilization (processors, memory, communications channels) should not exceed 50%.

Validation 3.

The standards consistent requirement for analyzability and behavior predictability places constraints on the selection of the implementation language. The language, or a predefined language subset, is required to have the following characteristics:

(a) a formally defined syntax,

(b) a means of enforcing the use of any subset,

(c) a well-understood semantics and a formal means of demonstrating equivalence of source code to the formal module design,

(d) block structured, and

(e) strongly typed.

At the present time requirement (c) can be met by languages which accept the Formal Specification of Modules (the Design) in terms of embedded mathematical assertions or annotations (pre and post conditions). As the code is developed to implement the design, new assertions about the state of variables can be determined at various "interesting" points such as entry into a loop construct. Eventually, a chain of assertions is developed. A semantic analyzer, such as Spade [21] or Gypsy [22] can automatically generate the Proof Obligations. Discharging the proof is then equivalent to demonstrating equivalence between the source code and the Design.

The standard requires that programs written in the selected language be analyzable by static path analysis tools. Static path analysis is a technique which operates on source code and can determine problem items such as:

(a) control flow anomalies, including unreachable loops, unreachable code, and loops with multiple exits,

(b) data flow anomalies, for example, the attempted use of unassigned variables or the non-use of variables which have been assigned a value,

(c) unwanted information flow dependencies (determinable by analyzing the relationship between input variable values and output variable values).

The static path analysis requirement means that languages containing constructs not amenable to static path analysis are prohibited. Many popular languages fall into this category including Ada and Pascal. This has stimulated the definition of language subsets in which unanalyzable constructs have been removed. Two well-known subsets are SPADE-Pascal [23] and SPARK-Ada [24].

As noted previously, the use of assembly language is acceptable, abut the standard suggests its use be minimized.

Again, to promote analyzability, the standard requires that structured programming be adopted as a programming standard.

Validation 4.

The standard employs traditional dynamic testing technology for the following illustrative reasons:

(a) Undetected logic errors may be present in the formal arguments. Such errors detected by dynamics testing are clearly serious and the standard states this would generally lead to non-acceptance of the SCS.

(b) Formal Arguments may not have been used in all cases.

(c) Validate conformance to non-functional requirements such as performance which are not usually the subject of formal arguments

The standard requires the use of random testing in addition to black-box testing. It is noted in the standard that random testing is a particularly economical means of generating large numbers of test cases. The black-box tests are compiled by the V&V Team who are also responsible for conducting the tests. At a minimum, all safety-critical features must be tested. An automated test harness equipped with a test coverage monitor is required. Tests on individual modules, except for I/O modules, will take place on the host (development machine) running a target machine emulator. I/O modules and integration testing are conducted on the target hardware.

Validation 5.

The final validation is the only part of the standard that involves a system level issue. The crucial role that SCS plays in the deployed system warranted attention at this point. The integrated hardware/software system is tested by the V&V Team to establish conformance (equivalence) to the original Software Requirements Specification. Prior to this time, all development equivalences and testing were along the right hand side of the triangle, i.e., the formalized side. As noted above, it is always possible that undetected errors remain on that path or that the initial animation of the formal specification was in some way incorrect . This last validation attempts to provide additional assurance that the system meets the intent of the English requirements specification.

It is apparent that the specificity of the requirements and the amount of guidance provided by the standard for the last two (testing) validations is far less than what is provided for the first three (formal) validations.

This concludes the discussion of existing military standards which focus on software safety matters. The next section discusses an open design issue concerning the isolation of safety-critical software.

7. Isolation of Safety-Critical Software

In regard to software safety, both of the standards we have examined have as their beginning points the identification of safety-critical software. Some have argued essentially that given an analysis which did in fact identify all safety-critical software, there is a non-zero probability that non-safety critical software could adversely impact the safety integrity of the system.

Parnas, et.al. [25], have argued that software often exhibits weak-link behavior; i.e., unanticipated relationships exist between seemingly uncoupled modules or data files. They suggest that safety-critical software be isolated on dedicated processors. Both standards require a separation of safety-critical (SCS) and non-safety critical (NSCS) software. The preference is separate machines followed by protected logical separation on one machine.

Addy [28] provides a detailed case study which supports the weak-link argument. He also included a discussion of the design tradeoffs involved in supporting isolation. Following is a summary of the tradeoffs.

Complete physical isolation on separate machines.

Advantages:

1. SCS would not share program control with NSCS software.

2. SCS would have exclusive use of its data.

3. SCS would not have to compete for resources or services.

Disadvantages:

1. Increased hardware costs.

2. Additional software development and maintenance costs (redundant interface modules, coordination between SC and NSC hardware.

3. Possibility of timing constraint violations.

Addy concluded that the additional costs and complexity make this first approach not feasible for most applications.

Separate Safety Process

Approach: dedicated memory is allocated to the safety process (SP). The SP maintains a separate database of safety-critical variables. Based on the value of these variables, the SP grants or withholds approval for safety-critical processing to begin. The monitored variables could be sensor input from the system environment, or status variables reflecting the state of the hardware or the integrity of software and data. This approach is based on the concept of a safety kernel, a monitoring and control mechanism analogous to security kernels in secure systems.

Advantages

1. A relatively small amount of code is needed to implement the SP. This facilities analyzability and predictability, leading to assurance.

2. Minimal change to existing code

Disadvantages

1. SP Software development costs
2. SP could become a performance bottleneck for safety-critical executions.
3. The SP is coupled to the other processes that provide it with data. Thus, the SP is less isolated than if the SP and its related processes were on a separate machine.
4. The authorized processes could cause corruption and bypass the SP.
5. Overall complexity is increased, particularly since the system requires an interface to communicate with the SP

No Isolation (Integration)

Advantages

1. Should lead to smaller total amount of code which generally impacts all concerns in a positive manner, particularly overall complexity.

Disadvantages

1. Budget considerations will require that safety-critical software be identified via analyses of the entire detailed design. Unknown interfaces between SCS and NSCS may remain undetected.

In the following initial work at defining a SAFETY quality factor, we have adopted the guidance of the standards that physical isolation of SCS and NSCS is required in the absence of compelling factors to the contrary.

8. A SAFETY Quality Factor

8.1 Definition

Five of the existing thirteen factors appear most appropriate on which to base a Framework definition of SAFETY. They are RELIABILITY, SURVIVABILITY, USABILITY, CORRECTNESS and VERIFIABILITY. Following is a discussion of these factors and their defining criteria within the context of software safety.

8.1.1 RELIABILITY

RELIABILITY measures freedom from software failure. The guidebook definition is: extent to which the software will perform without any failures within a specified time period. High reliability does not guarantee safety. While highly reliable software may have few failures, if a failure does occur which leads to a catastrophic event, the software has not met safety requirements and is fact not safe. To the extent that reliability is a part of AVAILABILITY (a system level factor which includes SURVIVABILITY and MAINTAINABILITY), SAFETY may be in conflict with RELIABILITY for two reasons.

For example, the presence of a safety interlock may interfere with the availability of some system capability. Another conflict would arise when the system fails to a safe state and all system capabilities are lost, this being an example of zero availability.

The defining criteria for RELIABILITY are:

accuracy, characteristics of software which provide the required precision in calculations and outputs,

anomaly management, characteristics of software which provide for continuity of operations under and recovery from non-normal condition, and

simplicity, characteristics of software which provide for definition and implementation of functions in the most noncomplex and understandable manner.

All three of these criteria seem relevant to safety-critical operations. V&V typically reviews accuracy of algorithms and output values. Safety features are very often designed to deal with anomalous circumstances. Simplicity supports analyzability, a high-level principle applied to safety critical software

8.1.2 SURVIVABILITY

SURVIVABILITY is the extent to which critical software functions will be supported when a portion of the system is inoperable.

The defining criteria for SURVIVABILITY, in addition to anomaly management, are:

autonomy, characteristics of software which determine its non-dependency on interfaces and functions,

distributedness, characteristics of software which determine the degree to which software functions are geographically or logically separated within the system,

reconfigurability, characteristics of software which provide for continuity of system operation when one or more processors, storage units or communication links fails, and

modularity, characteristics of software which provide well-structured, highly cohesive, minimally coupled software.

Autonomy is likely to be very important in a safety-critical system, since some percentage of safety critical software will interface with safety-critical hardware. If the safety-critical hardware fails, either stops functioning or produces incorrect outputs, the interfacing software, and possible the entire system may become corrupted or stop operating[17]. Depending on circumstances, safety features could fail.

Within the context of the software quality model, distributedness refers to the architecture or design structure of distributed systems. It is relevant to safety concerns to the extent it supports physical partitioning of SCCSCs and non-SCCSCs.

[17]My colleague, Guy Johnson, suggested the example of the Apple Macintosh mouse. Suppose the Mac controls a software-based safety feature which is initiated by operator control. Somehow, the mouse becomes disabled. The Mac is rendered inoperable and the safety feature cannot be invoked.

Reconfigurability is essentially an issue of fault-tolerance. It is relevant to safety concerns to the extent fault-tolerance is applied to safety-critical functions.

Modularity is a key software engineering design principle. In the context of safety-critical software, it supports the notion of logical partitioning of safety and non-safety critical software if physical partitioning is not feasible

In sum, autonomy and modularity are broadly related to software safety, compared to distributedness and reconfigurability which are more narrowly related

8.1.3 USABILITY

The guidebook definition of USABILITY is undesirable, being defined as the effort required to use the software relative to the effort required to implement the software. A more preferable definition would be some measure of ease of learning and of use with respect to the complexity or criticality of the functions provided. In the context of safety, high usability software would be safety-critical software designed with human factor considerations.

The defining criteria for USABILITY are:

operability, characteristics of software concerned with the usability of inputs and outputs to the software, as well as procedures for operating the software, and

training, characteristics of software which provide transition from current operation and provide initial familiarization. This guidebook definition is undesirable. A more preferable definition would be characteristics of the software which indicate the amount, type and duration of training required.

Both of these criteria are relevant to the operational aspects of safety-critical software.

8.1.4 CORRECTNESS

CORRECTNESS is narrowly defined to be the extent to which software design and implementation conform to specifications and standards. The three criteria of correctness deal exclusively with design and documentation formats and are content independent.

completeness, the indications that the required functions have been fully implemented,

consistency, the extent to which uniform design and notation are used, and

traceability, the indications that requirements have been fully allocated across several phases of the life cycle and across the corresponding several levels of documentation (not the guidebook definition).

Since they provide measures of compliance, these criteria help provide assurance that implemented safety-critical software meets all documented requirements.

8.1.5 VERIFIABILITY

VERIFIABILITY deals with software design characteristics affecting the effort to verify software operation and performance against requirements. Based on our analysis, two new criteria have been added to VERIFIABILITY, analyzability and predictability. In addition to the already defined criteria of *simplicity* and *modularity*, the defining criteria of verifiability are:

> *analyzability*, characteristics of software which allow or facilitate rigorous analysis,
>
> *predictability*, characteristics of software which provide assurance that execution results conform to requirements,
>
> *visibility*, reporting requirements that provide status monitoring of the software development with respect to verifiability, as opposed to for example, cost and schedule (definition adopted from guidebook),
>
> *self-descriptiveness*, characteristics of software which provide an explanation of the implementation of functions, and

According to the guidebook, visibility is concerned exclusively with the testing program. Self-descriptiveness is not fully implemented by the guidebooks, since the metrics for this criteria deal only with the establishment of code commenting standards. The scope of this criteria should be expanded to include detailed design. Simplicity of design and of implementation is good software engineering practice and is guidance contained in the standards reviewed above.

All of the criteria contribute to the essential activities of V&V analysis and review for safety critical software.

8.2 Metric Questions

This section contains candidate metric questions for the proposed factor SAFETY. The questions were derived from the investigation's background materials considered within the context of the Framework's existing quality factors and criteria.

The questions are mostly a synthesis of the concepts and requirements found in MIL-STD-882B Notice 1 and MoD 00-55/1. There is no intention to suggest that the two standards should be merged. Our aim in constructing the questions was to provide as broad a sample as possible. Thus, with a few exceptions, we did not repeat questions which are applicable to more than one development phase.

The questions are organized by criteria and then by the development level used by Guidebook Volume III [14]. In the Framework, questions are organized into worksheets (WS) Each worksheet corresponds to a development level. The five levels are System (WS 0), Software Requirements Analysis (WS 1), Preliminary Design (WS 2), Detailed Design (WS 3) and Code and Unit Test (WS 4). Worksheets 3 and 4 are each further divided into CSCI and Unit levels. Here, we did not distinguish between the CSCI and Unit levels.

There are only a few questions included for Code and Unit Test (Worksheet 4). In the case of software safety, the choice of implementation language will generate a large number of language-specific questions. For this initial effort, this appeared to be too specialized an area to include at this time. A good overview of the issues involved in selecting an implementation language for safety-critical applications is found in [30]. For an entire volume devoted to the use of Ada in safety-critical systems see [31].

METRIC QUESTIONS

ACCURACY (AC)

System Level (WS 0)

1. Have tolerances been established for safety-critical system inputs and outputs?

Software Requirements Analysis (WS 1)

1. Have tolerances been established for safety-critical software inputs and outputs?

ANALYZABILITY (AZ)

System Level (WS 0)

1. Is there a requirement that all specifications and designs be amenable to formal analysis.
2. Is there a requirement that all software shall be free of architectures and language constructs that would impede rigorous analysis of the software?
3. Have criteria been established for selecting the safety-critical software implementation language?

Software Requirements Analysis (WS 1)

1. Is there a requirement to use a standard subset of the implementation language for safety-critical software?
2. Is there a requirement to avoid or minimize the use of floating point arithmetic?
3. Is there a requirement to avoid or minimize the use of interrupts?
4. Is there a requirement to avoid or minimize the use of recursion?
5. Is there a requirement to avoid or minimize the use of parallel asynchronous processing?
6. Is there a requirement to avoid or minimize the use of multi-tasking?
7. Is there a requirement to avoid or minimize the use of object code patching?
8. Is there a requirement to avoid or minimize the use of dynamic program reconfiguration?
9. Is there a requirement to avoid or minimize the use of dynamic memory allocation?

Preliminary Design (WS 2)

1. Is the design amenable to formal proof or rigorous argument?

Detailed Design (WS 3)

1. (a). How many proof obligations?
   (b). How many proof obligations have been discharged?
   (c). Calculate b/a and enter score.

ANOMALY MANAGEMENT (AM)

System Level (WS 0)

1. Are there requirements for a safe survival and recovery from detected hazardous conditions (i.e., fail-operational)?

2. Are there requirements for continued safe system operation in degraded modes of operation (i.e, fail-soft)?

3. Are there requirements for the system to fail-safe?

4. Are there requirements for restarting the system from a fail-safe condition?

Software Requirements Analysis (WS 1)

1. Is there a requirement to use a safety-monitor?

Preliminary Design (WS 2)

1. Does the design incorporate capabilities for fail-operational, fail-soft, and fail-safe conditions?

AUTONOMY (AU)

System Level (WS 0)

1. Are there design requirements to isolate safety-critical software from faults or failures in safety-critical hardware?

Software Requirements Analysis (WS 1)

1. Is there a requirement for multiple copies of safety-critical software to protect against software corruption?

Preliminary Design (WS 2)

1. Does the design isolate safety-critical software from failures or faults in safety-critical hardware?

COMPLETENESS (CP)

System Level (WS 0)

1. Is the statement of each safety requirement specific and unambiguous?

2. (a). How many safety requirements?
   (b). How many safety requirements have been allocated to HWCI and CSCIs?
   (c). Calculate b/a and enter score.

3. Have all interfaces between safety-critical hardware and safety-critical software been identified?

4. Have requirements for appropriate system quality factors been specified for all safety-critical functions?

Software Requirements Analysis (WS 1)

1. Have all safety requirements been allocated to CSCIs?.

2. Have requirements for appropriate quality factors been specified for all safety-critical software functions?

Preliminary Design (WS 2)

1. Have all safety requirements been allocated to CSCIs and to CSCs?

2. Does the design incorporate all appropriate software quality factors?

Detailed Design (WS 3)

1. Have all safety requirements been allocated to the CSCI?

2. Does each allocated requirement meet the operational intent of the requirement?

Code and Unit Test (WS 4)

1. (a). How many units have been proved by formal arguments?
   (b). How many units proved by formal arguments have one or more errors?
   (c). if b/a = 0, enter 1; otherwise enter 0.

CONSISTENCY (CS)

System Level (WS 0)

1. Have criteria been established for formal design method selection?

Software Requirements Analysis (WS 1)

1. Is there a requirement to use a standard format for safety-critical software designs?

Preliminary Design (WS 2)

1. Are the design representations for safety-critical software in the format of the established standard

Detailed Design (WS 3)

1. Are the design representations for safety-critical software in the format of the established standard

Code and Unit Test (WS 4)

1. Have program constructs been reviewed to assure compliance with the subset implementation language?

DOCUMENT ACCESSIBILITY (DI)

System Level (WS 0)

1. Is documentation available which describes the results of previous Software Requirements Hazard Analyses which are applicable to the system?

2. Is documentation available which describes applicable safety standards and regulations?

3. Does the documentation specify the system's capabilities when operating in a safe state, a failed-soft state, and a failed-safe state?

4. Does the documentation contain comprehensive descriptions of all safety-critical functions?

Software Requirements Analysis (WS 1)

1. Has an initial set of safety-critical software components been identified?

2. Have all inputs to the safety-critical software been identified and documented?

3. Have all safe, hazardous and unsafe outputs from safety-critical software been identified and documented.

Preliminary Design (WS 2)

1. Does the documentation clearly show safety-critical software execution paths?

2. Does the documentation identify and state the purpose of the interface between each safety-critical CSC and safety-critical hardware?

3. Is there a table showing safety-critical CSCs which execute when the system is in a safe, failed-soft or failed-safe state?

Detailed Design (WS 3/3A)

1. Has documentation been developed for inclusion in the user's manual, operator's manual and diagnostic manual?

2. Is each formally specified design accompanied by an English commentary?

3. Does the documentation identify and state the purpose of the interface between each safety-critical CSU and safety-critical hardware?

3. Is there a table showing safety-critical CSUs which execute when the system is in a safe, failed-soft or failed-safe state?

MODULARITY (MO)

System Level (WS 0)

1. Is there a requirement to isolate safety-critical functions from non safety-critical functions

Software Requirements Analysis (WS 1)

1. Is there a requirement to isolate safety-critical software from non safety-critical software?

2. Is there a requirement to isolate safety-critical data from non safety-critical data?

Preliminary Design (WS 2)

1. Has all safety-critical software been isolated from non safety-critical software?

2. Has all safety-critical data been isolated from non safety-critical data?

OPERABILITY (OP)

System Level (WS 0)

1. Are there requirements to alert the user/operator of safety-critical software error and safety-critical equipment malfunctions?

2. Are there requirements to provide the user /operator with the capability to cancel a process or event?

Software Requirements Analysis (WS 1)

1. Are there requirements to provide the user/operator unambiguous and complete display of safety critical information

Preliminary Design (WS 2)

1. Does the design incorporate human factors considerations for safety-critical software functions which interface with the user/operator?

PREDICTABILITY (PR)

System Level (WS 0)

1. Is there a requirement that the behavior of the system be predictable at all times?

Software Requirements Analysis (WS 1)

1. Is there a requirement that the programming language manual be free from reference to undefined (i.e., unpredictable) results?

2 Is there a requirement to use a validated compiler for safety critical software?

3. Is there a requirement to limit peak safety-critical utilization to 50%.

4. Is there a requirement for the use of defensive programming?

5. Are there requirements for meeting hard real-time constraints for safety-critical software?

Preliminary Design (WS 2)

1. Does the design incorporate run-time assertions for safety-critical software?

2. (a). What is the total hardware resource allocated?
   (b). What is the estimated hardware resource used by safety-critical software?
   (c). Calculate b/a; enter 1 if b/a <= 50%, otherwise enter 0.

Detailed Design (WS 3/3A)

2. (a) How many CSUs.
   (b) How many CSUs with defensive programming?
   (c) Calculate b/a and enter score.

1. (a) How many units?
   (b) How many units check data inputs from other software units?
   (c) Calculate b/a and enter score

2. (a) How many units?
   (b) How many units check data inputs from other hardware units?
   (c) Calculate b/a and enter score

RECONFIGURABILITY (RE)

System Level (WS 0)

1. Is there a requirement for safety-critical features to be fault-tolerant?

Software Requirements Analysis (WS 1)

1. Are there requirements for safety-critical software to be fault-tolerant?

Preliminary Design (WS 2)

1. Does the design include redundant safety-critical software?

2. Does the design incorporate features for maintaining the integrity of all safety-critical data values following the occurrence of anomalous conditions?

3. Does the design include redundant safety-critical data?

SELF-DESCRIPTIVENESS (SD)

System Level (WS 0)

1. Is there a requirement for each formal design to be accompanied with an English annotation

SIMPLICITY (SI)

System Level (WS 0)

1. Is there a requirement to develop all safety-critical software in a way which facilitates understanding of the design and program structure?

2. Is there a requirement to develop all safety-critical software according to structured design techniques

Software Requirements Analysis (WS 1)

1. Is there a requirement to avoid or minimize the use of assembly language?

TRACEABILITY (TC)

System Level (WS 0)

1 Has a Safety Requirements Traceability Matrix (SRTM) been created?.

2. (a). How many safety requirements?
   (b). How many safety requirements have been recorded in the SRTM ?
   (c). Calculate b/a and enter score.

3 (a). How many safety requirements?
   (b). How many safety requirements have test references recorded in the SRTM?
   (c). Calculate b/a and enter score.

Software Requirements Analysis (WS 1)

1. Have all safety requirements allocated to software been recorded in the SRTM.

Preliminary Design (WS 2)

1. Has all safety-critical software been placed under configuration control?

Detailed Design (WS 3/3A)

1. Have all new SCCSCs been placed under configuration control?

TRAINING (TN)

System Level (WS 0)

1. Are there requirements to provide training materials describing the system's safety features and safe operation?

2. Are there requirements to provide realistic simulation exercises of hazardous conditions?

Software Requirements Analysis (WS 1)

1. Are there requirements to provide training materials describing the system's software-controlled safety features?

VISIBILITY (VS)

System Level (WS 0)

1. (a). How many safety-critical requirements?
   (b). How many safety requirements have preliminary qualification tests?
   (c). Calculate b/a and enter score.

Software Requirements Analysis (WS 1)

2. Is there a requirement to design tests specifically for safety-critical software

Preliminary Design (WS 2)

1. Have test requirements been specified for safety-critical CSC integration and test?

Detailed Design (WS 3/3A)

1. Is there a test for each safety requirement?

Code and Unit Test (WS 4/4A)

1. Has every safety requirement been tested?

9. REFERENCES

[1] Michael J. D. Brown, "Rationale for the Development of the UK Defence Standards for Safety Critical Software," *COMPASS '89 Proceedings of the 5th Annual Conference on Computer Assurance*, IEEE, 1990, pp.144-150.

[2] Department of Defense, MIL-STD-1574A (USAF), *System Safety Program for Space and Missile Systems*, 15 August 1979.

[3] Department of Defence, MIL-STD-882B Notice 1 *System Safety Program Requirements*, 1 July 1987.

[4] Ministry of Defence, Interim Defence Standard 00-55/1, *The Procurement of Safety Critical Software in Defence Equipment, Part I: Requirements*, 5 April 1991. The standards are available from Ministry of Defence, Directorate of Standardization, Kentigen House, 65 Brown Street, GLASGOW G2 8EX, ENGLAND.

[5] Ministry of Defence, Interim Defence Standard 00-55/1, *The Procurement of Safety Critical Software in Defence Equipment, Part II: Guidance*, 5 April 1991.

[6] Ministry of Defence, Interim Defence Standard 00-56/1, *Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment*, 5 April 1991.

[7] Department of Defense, *MIL-STD-SNS (Navy), Software Nuclear Safety.*

[8] Michael L. Brown, "Software Systems Safety and Human Errors," *Compass '88 Proceedings of the 3rd Annual Conference on Computer Assurance*, IEEE, 1988, pp. 19-28.

[9] M.A. Ould, "Software development under Def Stan 00-55: a guide," *Information and Software Technology*, (32,3), April 1990, pp. 170-175.

[10] Nancy G. Leveson, "Software Safety: Why, What and How," *Computing Surveys*, (18,2), June 1986, pp. 125-163.

[11] Nancy G. Leveson, "Safety as a Software Quality," *IEEE Software*, May 1989, pp.88-89.

[12] Galen Gruman, "Software Safety Focus of New British Standard," *IEEE Software*, May 1989, pp. 95-96.

[13] Department of Defense, DoD-STD-2167A, Defense System Software Development, 29 February, 1988.

[14] Thomas P. Bowen, Gary B. Wigle and Jay T. Tsai, "Specification of Software Quality Attributes," RADC-TR-85-37, Final Technical Report, February, 1985.

[15] Nancy G. Leveson and Peter R. Harvey, "Analyzing Software Safety," *IEEE Transactions on Software Engineering*, (SE-9, 5), September, 1983, pp. 569-579.

[16] Nancy G. Leveson and Janice L. Stolzy, "Safety Analysis Using Petri nets," *IEEE Transactions on Software Engineering*, (SE-13, 3), March, 1987, pp. 386-397

[17] Janet A. Gill, "Software Safety Analysis in Heterogeneous Multiprocessor Control System," Proceedings Annual Reliability and Maintainability Symposium, Orlando, IEEE, 1991, pp. 290-294.

[18] Department of Defense, *U.S. Air Force Software Safety Handbook.*

[19] Cliff B. Jones, *Systematic Software Development Using VDM 2nd Edition*, Prentice-Hall, 1989.

[20] Antoni Diller, *Z: An Introduction to Formal Methods*, John Wiley, 1990.

[21] B.A. Carre, et. al., "SPADE - The Southhampton Program Analysis and Development Environment," in I. Sommerville (ed.), *Software Engineering Environments*, Peter Pereginus, 1986.

[22] Allen L. Ambler, et. al., "GYPSY: A Language for Specification and Implementation of Verifiable Programs," in *Software Specification Techniques*, Addison-Wesley, 1986.

[23] B. P. Philips and S. G. Howe, "Verification - The Practical problems," in B. K. Daniels (ed.), *Achieving Safety and Reliability in Computer Systems*, Elsevier Applied Science, The Netherlands, 1987, pp. 89-99.

[24] B. A. Carre and T. J. Jennings, *SPARK - The SPADE-Ada Kernel*, University of Southhampton, UK, 1988.

[25] David L. Parnas, A. John van Schouwen and Shu Po Kwan, "Evaluation of Safety-Critical Software," *Communications of the ACM*, (33,6), June 1990, pp. 636-647.

[26] Bishop Brock and Warren A. Hunt, Jr.,"Report on the Formal Specification and Partial Verification of the VIPER Microprocessor," *Compass '91 Proceedings of the Sixth Annual Conference on Computer Assurance*, IEEE, 1991, pp. 91-98.

[27] R. A. Gove and Janene L. Heinzman, "Safety Criteria and Model for Mission Critical Embedded Software Systems," *Compass '91 Proceedings of the Sixth Annual Conference on Computer Assurance*, IEEE, 1991, pp. 69-73.

[28] Edward A. Addy, "A Case Study on Isolation of Safety-Critical Software," *Compass '91 Proceedings of the Sixth Annual Conference on Computer Assurance*, IEEE, 1991, pp. 75-83.

[29] Department of Defense, *Trusted Computer System evaluation Criteria*, DoD-STD-5200.28, 15 August 1983.

[30] W. J. Colleyer, S. J. Goodenough and B. A. Wichmann, "The Choice of Computer Languages for use in Safety-Critical Systems", *Software Engineering Journal*, March, 1991, pp. 51-58.

[31] I. C. Pyle, *Developing Safety Systems: A Guide Using Ada*, Prentice Hall International, 1991.

SECTION IV

OBJECT-ORIENTED TECHNOLOGY and SOFTWARE QUALITY

1. INTRODUCTION

Since 1976, RL has been funding the development of a software quality measurement methodology. During this time and up to the present, this research program has been the only sustained effort which has attempted to define a methodology to specify and evaluate software quality at the product level. The current definition of the methodology is found in a three volume set of guidebooks, *Specification of Software Quality Attributes* [7]. The key underlying concept of the methodology is a three level hierarchical model of software quality . The top level is a set of thirteen customer-oriented *software quality factors*. The factor set is Efficiency, Integrity, Reliability, Survivability, Usability, Correctness, Maintainability, Verifiability, Expandability, Flexibility, Interoperability, Portability, and Reusability.

The second level is a larger set of defining attributes for the software quality factors. These are termed *software quality criteria* and reflect technical considerations of good software engineering and development practice. The third level is a still larger set of *software quality metrics* which are measures of the software quality criteria. The software quality metrics in turn are defined by lower level metric elements.

Rome Laboratory intends to periodically update the Software Quality Measurement methodology in order to incorporate changes in software engineering technology. This effort is an initial attempt to incorporate object-oriented technology into the methodology.

The 1980s witnessed rapidly increasing interest in object-oriented technology, methods, and concepts. The object-oriented approach stands in contrast to traditional methods based on functional decomposition. Advocates of this approach claim that by concentrating on the objects comprising a system, rather than the functions it performs, one arrives at clearer specifications, cleaner designs, and more maintainable implementations.

One problem, however, is that existing methods of assessing design and implementation quality are inadequate for evaluating object-oriented artifacts. In particular, existing metrics based on functional decomposition are not always appropriate (or even meaningful) when applied to object-oriented software development, and even where common terminology is involved, subtle distinctions often give rise to widely varying assessments of quality. Unfortunately, the continuing rapid development of object-oriented methods has resulted in scant attention being paid to the assessment of software quality. It is the goal of this work to present some preliminary metrics that, based on current accepted criteria for "good" design, will aid in the evaluation of future design. However, the technology is still young, and this work must be viewed as an initial contribution only. As the field matures, the proposed metrics should be reviewed and modified, replaced, or enhanced as necessary in light of advances in research and application.

Of course, the proposed metrics cannot be comprehended or applied in a vacuum. Thus, another goal of this work is to provide the necessary background in the state of the art in object-oriented development as a prelude to the discussion of object-oriented quality concepts and methods. To meet these objectives, the remainder of this paper is organized as follows:

- Section 2 provides background information related to historical development of object-oriented technology, as well as the key concepts underlying the application of this technology to software development. Included is a brief discussion of the benefits that proponents claim accrue to the use of this technology.

- Section 3 expands the previous discussion by considering the specific application of object-oriented technology to software design. This includes a discussion of design issues and emerging design disciplines as these relate to software quality. Much of this work is subjective in nature, and not directly transferable into metrics. However, these subjective criteria are the basis for our proposed metrics.

- Section 4 presents the proposed quality metrics for object-oriented design and implementation within the framework described in [7].

2. KEY CONCEPTS

Any quality evaluation system requires an understanding of the key concepts and assumptions underlying the development methods employed. This section addresses this issue in the context of object-oriented development The first subsection provides a historical perspective on the development of the object-oriented paradigm. The remaining subsections provide a short introduction to key concepts of object-oriented technology. Of course, this brief introduction is insufficient for a deep appreciation of the issues involved; references [6, 31, 54] have excellent extended discussions of the technology.

2.1 Historical Development

Most proponents of object-oriented technology trace the development of this technology back to the introduction of Simula 67 [12]. Simula 67 introduced many of the constructs considered essential to object-oriented programming:

- Objects comprising state (variables) and operations (procedures to access and modify the state).

- Classes categorizing the distinct types of objects in the system.

- Inheritance relating groups of classes by generalization/specialization relationships.

- Polymorphism allowing objects to be used in any context where an object of a ancestor class is required.

- Dynamic binding permitting specialized classes to extend the operations defined in their ancestor classes.

Although originally developed for simulation (hence its name), Simula 67 quickly attracted a loyal (if small) group of admirers who saw the object classification paradigm as a powerful mechanism for the organization of large software systems. The work of the Simula 67 group, as well as that of Hòare on data types [19] and the development of languages such as Pascal [55] led to two divergent lines of research in the 1970s: abstract data types (ADTs), and object-oriented programming.

Work on abstract data types was most visible, resulting in a succession of languages emphasizing data abstraction: Euclid [25], CLU [28], Modula-2 [56], and Ada [48]. For the most part, the design and development philosophy underlying these languages was that, while ADTs extended the types of entities which a software system could manipulate, these were passive elements identified and designed fairly late in the software development cycle. The primary use of ADTs was to enhance, rather than replace, conventional functional decomposition. In particular, data type design was usually of secondary importance to functional design.

A few groups, however, continued to view objects as the primary concepts behind software construction. The most influential group was that at Xerox PARC, where a succession of Smalltalk languages was developed. Much of this work was underpublicized until the start of the 1980s with the announcement and release of Smalltalk 80 [16]. In parallel with this, researchers elsewhere were reviving interest in object-oriented technology by extending conventional languages with support for classes and inheritance. The most popular extension languages were C [21] and LISP [44]. The former was the base for the development of both C++ [47] and Objective-C [11], while the latter resulted first in the Flavors package [32], and most recently in the Common Lisp Object System (CLOS) [3].

The increased interest in object-oriented programming led to the development of new languages such as Eiffel [30], that attempt to combine the pure object-oriented approach of Smalltalk 80 with facilities to support high quality, software development. In addition, others have attempted to extract the essence of the object-oriented paradigm and use this to synthesize complementary methods for analysis and design [6, 10, 39,54]. The current state-of-the-art is that a variety of languages exist to support object-oriented development, and disciplines are emerging to support such development in up-front tasks such as analysis and design.

2.2 Objects and Classes

The three most important concepts underpinning object-oriented development are objects, classes, and inheritance. The goal of this subsection is to define these terms and briefly show how they serve *in toto* to define the object-oriented paradigm.

2.2.1 What is an Object?

As might be expected, the notion of what constitutes an object is at the core of any method, technology, or language that claims to be object-oriented. While practitioners and theoreticians disagree as to details, in general an object is viewed as a self-maintaining and self-monitoring entity of interest in the problem domain. Along one dimension of organization, an object may be a software model of a concrete entity: a person, aircraft, or sensor, or it may represent an abstract concept, such as an airline flight, a passenger list, or a graphical window. Orthogonal to this dimension are the notions of objects representing self-contained, indivisible entities, such as persons or sensors, and those objects representing collections of other objects (e.g., arrays, lists, stacks, and queues).

Various writers have attempted to describe the essence of an object. For instance, Wegner [50] writes "An object is a set of 'operations' and a 'state' that remembers the effects of these operations.". Booch, in his recent text on object-oriented design [6], states that "An object has state, behavior, and identity." The common ground here is on an object as an encapsulation of useful state information, and one can view the operators of Wegner as defining the behavior emphasized by Booch.

Simply put, each object is in control of its own destiny. This anthropomorphism is actually a key to understanding the object-oriented viewpoint. Another indicator of this viewpoint is the use of *message* in Smalltalk [16] to denote the request for an operation on an object (whether this is to interrogate the object's state or to alter its state). While some have objected to this term as implying a (non-existent) parallelism in Smalltalk (by analogy to message passing in operating systems), the terms does connote that objects receive messages as requests to perform some action. This in turn reinforces the identity and inviolability of each object's internal state. As Ingalls has so succinctly phrased it:

> Instead of a bit-grinding processor raping and plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires [20].

Of course, one must ascribe positive attributes to this viewpoint in order to convince others of its viability and merit. One immediate consequence of the object-oriented viewpoint is that it enhances the properties of abstraction and information-hiding. The former is enforced by the encapsulation of all state information within an object, and thus forbidding direct manipulation by other unrelated objects. That is, the behavior of an object is fully specified by the operations forming its public interface. Similarly, information hiding is also enforced by encapsulation. Other objects have no way of knowing what they do not need to know, just exactly *how* an object implements the services it provides. This permits the replacement of one object by another whose behavior is identical, but whose internal data structures and algorithms may be radically different. This makes it feasible to define behavior more formally, via a state invariant that the object must ensure, as well as pre- and post-conditions defining the effects of each operation.

Objects, then, represent an alternative structuring mechanism to those based on functional decomposition. By consolidating the operations and state information of a model entity in an object, one can attain many of the attributes indicative of well-engineered software; i.e.,.abstraction, encapsulation, and information hiding. However, the existence of objects leaves open the question as to how an object's state and behavior are defined. Providing these definitions is the role of the two other primary object-oriented concepts: classes and inheritance.

2.2.2 What is a Class?

It is rare indeed that the individual entities in a software system are entirely unrelated in form or function. Indeed, one of the key issues in software design is the identification of appropriate abstractions that generalize particular behavior. Traditionally, these abstractions have been functional in nature. There may be several routines, for instance, that provide the sorting abstraction in a variety of algorithms. With object-oriented technology, one abstracts whole entities that in turn define the system model.

In general, there will be many objects that, in the model at least, exhibit common behavior. For example, all queues exhibit FIFO behavior, whether queues contain operating system I/O requests, customers at a bank, or mouse events in a window manager. Similarly, all employees being modeled for a payroll system exhibit similar behavior, and maintain similar state information. Of course, each individual will have, in general, a distinct state, but the behavior with respect to that state is common.

The traditional way to define common behavior is via a *type*, and in the case of designer or programmer created types, an *abstract data type* (ADT). In object oriented methods, this notion is subsumed under the notion of a *class*. While classes in most object-oriented languages are subtly different from types (see section 2.3.1), for the present we will consider the concepts' similarities. An ADT defines the behavior of a particular variable type, whereas a class defines the behavior of a group of related objects. What matters here is the difference in perspective. Objects are modeled as self-contained, entities in charge of their own destinies, whereas variables of an ADT are traditionally viewed as being subservient to the functional processes in which they (the ADTs) are embedded.

While details vary, typically class definitions contain declarations for the variables defining the state of the objects in the class *instance variables*, as well as the signatures and implementations for the object operations forming the interface. In Smalltalk terminology, the signature defines the message protocol and the message name along with the number of arguments, and possibly their types. The implementation defines the *method* that provides the operation. Of course, at the specification and design stages, the code is replaced by a more or less formal description of the operation's effect. (In this regard, Eiffel [30] is notable for its inclusion in the language of a simple assertion mechanism to define preconditions and postconditions for each operation, as well as a state invariant that each object in the class must maintain.)

The mechanism for creating objects and associating them with identifiers varies widely. In Smalltalk, each class defines a *new* method to create properly initialized objects in the class. Indeed, as classes themselves are objects in Smalltalk, the result is a clean, self-referential system, at the cost, however, of maintaining type information during execution.

In Eiffel and C++, classes as such do not exist at runtime. Instead, special procedures are provided to create and initialize new objects. Conceptually, these are part of the object's behavior (as opposed to the Smalltalk approach of factoring these out into separate Class objects). As Eiffel and C++ are statically typed, there is no ambiguity as to the class of each object being created at any given instant, so the class protocol of Smalltalk (with its associated overhead and runtime type checking) is eliminated. As a general rule, embedded production level software developed using object-oriented techniques will be implemented in languages such as Eiffel and C++, both for the efficiency and higher reliability associated with static typing. We will continue to discuss Smalltalk, however, in that it is often the purest expression of some concepts that are necessarily constrained in statically typed languages.

In summary, a class serves to defined the (encapsulated) state of a related group of objects, and to specify the externally observed behavior via the operation interfaces. The next step is to handle the controlled evolution and specialization of systems in this context. The primary object-oriented concept addressing these needs is inheritance.

2.3 Inheritance

Classes by themselves are useful. Classes coupled with inheritance, however, provide a powerful mechanism for exploiting similarity of behavior, by creating a hierarchy of behavior based on these similarities. Note that classes by themselves have a rigid definition of similarity: two objects are similar if their behavior in the same state is identical, and they represent this state in the same manner. Inheritance supports incremental evolution based solely on behavioral similarity.

When a new class is created, it will typically reference a class from which it inherits both state and behavior. For example, classes defining polygons and ellipses might inherit from the class of closed figures, and the class of parallelogram objects could inherit from polygon (see Figure 1).

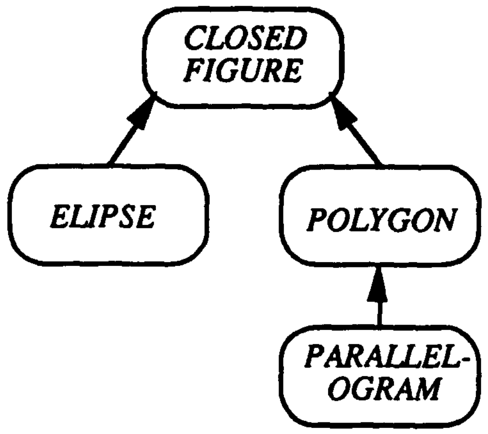


Figure 1: Simple Inheritance Structure

In such a case, class *CLOSED FIGURE* would be called the *superclass* of *POLYGON* and *ELLIPSE*, and, by symmetry, the latter two classes would be called *subclasses* of *CLOSED FIGURE*. This superclass/subclass organization defines a class hierarchy, and we (recursively) define the *ancestors* of a class *C* to be the superclass of *C* and the ancestors of *C's* superclass. In a similar manner, we can (recursively) define a class's *descendants* or *heirs* in terms of the class's subclasses.

Typically, a subclass has access to internal details of its superclass that are not part of the interface presented to most clients. In particular, the subclass can often directly manipulate the superclass instance variables and override the superclass methods. This latter property is especially powerful, as it means that a subclass can take advantage of special properties to optimize or specialize the behavior specified by the superclass. For example, rotating an arbitrary closed figure about its center is quite complex, whereas rotating an ellipse is simple by comparison. Thus the *ELLIPSE* class in Figure 1 might override the general method with a more efficient one based on the properties of ellipses. Indeed, were we to create a subclass *CIRCLE* of *ELLIPSE*, we could again override the method again with the simplest one possible, a method that does nothing!

Another use of inheritance is to extend the behavior of the superclass. For example, we might query a polygon object as to its number of sides, whereas this is meaningless for ellipses, as well as for the more general class of all closed figures. Thus, we might add a method in *POLYGON* to return the number of sides. In cases where we know we have a *POLYGON* we can send the associated message. However, *POLYGON*s, *ELLIPSE*s, and all of their subclasses are also *CLOSED FIGURE*s, so we can be confident that they will all respond appropriately to a common message to compute their area (even if the methods to perform the computation vary from class to class).

Some object-oriented languages have a special designated class that is the common ancestor of all other classes: in Smalltalk this is class *Object*, and in Eiffel this class is named *ANY*. Other languages, especially hybrids such as C++, do not define any fixed common ancestor class. In those languages that support this concept, messages available in all classes (such as a generic print message) will be implemented in this top-level class.

The issues involved in the use of inheritance are both important and subtle. The following subsections discuss various aspects of the use of inheritance in design and implementation.

2.3.1 Inheritance and Data Typing

It is easy but misleading to equate classes and subclasses with types and subtypes. While inheritance can be used to create a compatible type hierarchy, this is not a necessity. Indeed, some languages such as POOL [2] make clear linguistic distinctions between these two concepts. This subsection will discuss those cases where inheritance is used for subtyping, whereas the next subsection will concentrate on inheritance for code reuse.

When inheritance is used for typing, then we are using the class hierarchy to represent an "is-a" relation among classes. Referring again to Figure 1, we see that this relation is obeyed: a *PARALLELOGRAM* is a *POLYGON*, a *POLYGON* is a *CLOSED FIGURE*, etc. The key principle here is substitutability: can an object be used in any context where an object from one of its ancestor classes is required? If so, we have behavioral compatibility and by extension a proper type hierarchy.

Given a class hierarchy that is also a type hierarchy, there are two ways in which a class *C* may be related to its superclass. The first is by *type extension*:. Objects of class *C* extend the behavior of the superclass by adding additional methods, while maintaining compatibility with the methods defined in the superclass. Adding a method to compute the number of sides to the *POLYGON* class is an example of extension.

The second relationship is that of concrete implementation to abstract specification. Consider Figure 2, which shows three classes supporting a last-in first-out stack.

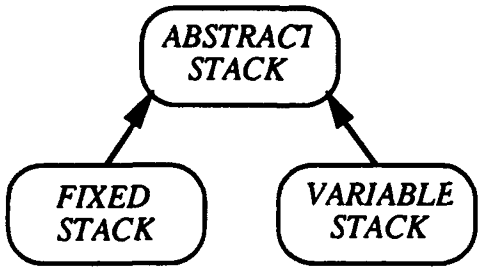


Figure 2: Abstract and Concrete Classes

At the top of the hierarchy is class *ABSTRACT STACK*, which defines the protocol common to all stacks (i.e., defines the appropriate messages and their arguments). However, this class may well *not* provide full implementations of the associated methods, leaving this chore to its subclasses. Such classes--defining behavior but omitting concrete implementations for one or more methods--are normally called *abstract classes* (or, in Eiffel, *deferred classes*).

Abstract classes are similar to package specifications in Ada, but they have one important difference. Some of the methods may be fully defined in terms of messages that must be implemented in subclasses. For example, *ABSTRACT STACK* may specify two messages, one to determine the "size'" of the stack, and the other to determine whether or not the stack is "empty". Obviously, the "empty" method can be written in terms of "size", even though the method for determining the stack size is unimplemented.

Continuing with the example, one can envisage the development of two concrete implementations of stacks: *FIXED STACK* using arrays, and *VARIABLE STACKs* built with linked lists. Both of these have *ABSTRACT STACK* as the superclass, and both *exactly* implement the defined behavior. Thus, objects from either concrete class can be used wherever a stack *per se* is required, without requiring any special knowledge on the part of the stack user. Note as well that neither concrete implementation is required to implement the "empty" method, though this is possible if desired.

2.3.2 Inheritance and Code Reuse

A second and more controversial use of inheritance is for code reuse. Often a c[illegible] will define useful behavior that is similar (but not identical) to that desired. For instance, a queue is very similar to a general list except for the rules governing adding and removing elements. If the existing *LIST* class also provides useful messages that traverse the list as a whole, it is tempting to have the new class *QUEUE* be an heir of *LIST*, overriding the insert and delete methods with ones appropriate to a queue. In this way, the *QUEUE* class can directly reuse a great deal of useful code from LIST. Unfortunately, the principle of substitution is violated, as queues cannot in general be substituted wherever a list is required.

Some other more recent languages, notably POOL [2], have separated the notion of type and class. In this way, inheritance is used to capitalize on reuse possibilities, while behavioral specifications define the type hierarchy. That is, classes and inheritance define the behavior objects possess, while types and subtypes define the behavior variables and parameters required. Assignment is only permitted when these behaviors match. In general, a class is not type-compatible with its superclass, though of course such compatibility is still possible.

Another approach is found in the most recent versions of C++. If class *C* lists class *B* as a superclass, then objects of class *C* can be substituted wherever an object of class *B* is required; that is, normally the class and type hierarchies are the same. However, if *C* lists *B* as a **private** superclass, then objects of class *C* have access to the state and methods of *B* (as would any subclass of *B*), but objects of class *C* are *not* substitutable for *B's*.

2.3.3 Multiple Inheritance

The inheritance hierarchy described so far is tree-structured, every class having at most one superclass. However, there are instances where the class being described has an "is-a" relation with several existing classes. Consider the extension to the class of figures shown in Figure 3.

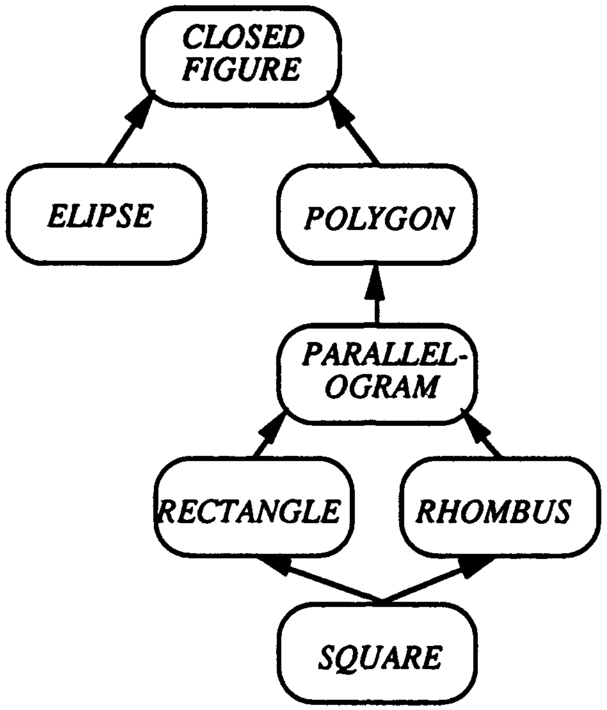


Figure 3: Example of Multiple Inheritance

New classes *RECTANGLE* and *RHOMBUS* are subclasses of *PARALLELOGRAM*, and class *SQUARE* has been made a subclass of both *RECTANGLE* and *RHOMBUS*. In a mathematical sense, this is appropriate, as squares have exactly the properties of both rhombuses and rectangles.

There are subtle and difficult problems with multiple inheritance. The first is method selection. If a class such as *SQUARE* does not implement a given method, where should the method be located: in *RECTANGLE*, in *RHOMBUS*, or in both? Also, all *SQUARES* are descendants of *PARALLELOGRAM* along two different paths. Should the instance variables for *PARALLELOGRAM* be duplicated, or should there be only a single copy?

Different languages handle these problems in a variety of ways. Suffice it to say that multiple inheritance is still an area of active research, and no generally acceptable solution to the problems it creates has been proposed. Indeed, some in the field believe that multiple inheritance is a red herring, and that any problem for which multiple inheritance is proposed can be solved as easily using only single inheritance. Given the widely different semantics attributed to this concept, it is a good heuristic to avoid the use of multiple inheritance whenever possible.

2.3.4 Clients vs. Heirs

Inheritance gives rise to two distinct ways in which different classes may be related. The first is a client/server relation, where objects in the first class have instance variables containing objects in the second class. The second relation is inheritance, where objects in the first class, being descendants of the second class, have access to state information not visible to normal clients. The obvious question is how to choose between these two.

The standard answer is to ask whether the "is-a" relation holds. If so, inheritance is indicated. Otherwise, the relationship is one of "has-a" or "contains," in which a client/server relation is more appropriate. In addition, if both of these appear appropriate, most designers and programmers would choose the client/server relation, as this results in the least coupling between the classes. As in traditional functional decomposition, lower coupling is usually an indicator of an artifact that is easier to maintain, test, and modify.

2.4 An Alternative: Prototypes and Delegation

The primacy of classes and inheritance as the structuring mechanism for objects is *not* universally agreed to by workers in the field. Languages such as Actor [1] and Self [49] do not have classes, nor do they support inheritance. Instead, these languages have a single entity, the *object*, as the basis for all internal structure. However, both Self and Actor embody concepts that mirror the effects of classes and inheritance, namely *prototypes* and *delegation*.

Prototyping is associated with new object creation. Instead of creating an object by mentioning the class to which it belongs, the object is created by cloning (copying) an existing object. In the process of making the copy, new instance variables and methods can also be defined, so the activity may resemble both object and subclass creation.

Inheritance is achieved by delegation. An object can specify that any message which it does not understand directly is to be passed to a *parent* object for processing. The similarity to subclassing and dynamic binding is obvious, and delegation can be used to simulate the inheritance mechanism found in most object-oriented languages.

While this approach is interesting, and is particularly attractive in rapid prototyping situations, it is not in the mainstream of object-oriented techniques, nor is it found in languages or methodologies whose focus is high reliability embedded systems. Thus, for the remainder of this paper, we will concentrate on the use of classes and inheritance to describe the behavior of similar objects.

2.5 Other Issues

In addition to the basic concepts of object, class, and inheritance, other issues are pertinent to the development of object-oriented software. This section briefly reviews a few of these issues.

2.5.1 Polymorphism

The term polymorphism, meaning "multiple structure" was coined by Strachey [46] to describe functions that operate on arguments of different types. Strachey was particularly interested in parametric polymorphism, where the function was defined uniformly for a given set of types, each of which was required to exhibit a given form of behavior. In this respect, parametric polymorphism is similar to the use of generic type parameters in Ada programs.

However, inheritance provides examples of another form of polymorphism based on the substitution principle. That is, if we assume the inheritance structure implements a type hierarchy, then we can substitute subclass objects wherever a member of the superclass is required. In particular, in strongly typed languages, this permits us to assign an object to a variable whose type is that of the object or any of its ancestor classes. Of course, by assigning to a variable of an ancestor class type, any extended behavior of the objects "true" class is hidden. We implicitly used this previously when discussing the substitution of *ELLIPSE* objects for a *CLOSED FIGURE* was required.

Thus, it is possible to have strong typing in place, and yet permit a variable to be associated with objects of many different types (any subclass of the variable's nominal class). Languages like Eiffel make extensive use of inheritance polymorphism to provide flexibility in the face of static type checking.

2.5.2 Dynamic Binding

A related concept is that of *dynamic binding*, which most frequently comes into play when refining abstract classes. Recall that the abstract class *A* simply defines the behavior expected when a given message is received, but need not provide an effective method to handle the message. Thus, if we have an object of class *A* in hand, any such message will be handled differently in different concrete subclasses.

Refer once again to the stack example in Figure 2. If we send a "pop" method to an *ABSTRACT STACK* object, the actual method will depend on whether the stack is actually a *FIXED STACK* or a VARIABLE STACK. In the former case, we would expect an array index to be decremented, whereas in the latter case a linked list node would probably be freed. The key observation is that either approach correctly implements a stack behavior, so objects from either concrete subclass may be freely used wherever an *ABSTRACT STACK* is required. The method appropriate to the specific object at hand will be selected dynamically.

It should be noted that dynamic binding is not restricted to implementing abstract messages only. It is also possible to override a fully-implemented method in a superclass if the attributes of a subclass permit optimization. Refer to Figure 3 again, and consider a message defined in *CLOSED FIGURE* to compute the internal area. At the level of *CLOSED FIGURE* this message would be specified but not implemented. Within *POLYGON* we might find a general method applicable to all polygons. However, this general method can be optimized in the case of *RECTANGLE*s, and even further simplified for objects in class *SQUARE.* An object in any descendant class of *CLOSED FIGURE* will handle the "area" message in a manner most appropriate to the object. This will occur even when the type of the variable associated with the object is only guaranteed to be *CLOSED FIGURE.*

Dynamic binding is primarily an implementation concept, as during design the concern is with observable behavior and not the specific algorithm that provides the behavior. However, dynamic binding is a key ingredient in supporting code reuse and software evolution, as it supports the incremental adaptation of a system to new demands without rewriting significant pieces of code.

2.5.3 Object-Oriented vs. Object-Based Languages

The previous discussion has stressed the use of inheritance, especially as a tool for behavioral extension, as well as a mechanism for providing concrete implementations of abstract classes. However, many common languages do not support inheritance, or support it in a very restricted sense, yet do have constructs analogous to class and object. Chief among these is Ada, where packages and private types can be used to define classes and create objects. However, the lack of inheritance in the current version of Ada leads most workers to classify it as *object-based* rather than *object-oriented* [50].

This is not simply semantic quibbling. Many of the object-oriented design approaches emphasize the need to identify a hierarchy of classes as part of the overall design. Clearly, implementing such a design in a language like Ada imposes serious problems, as there are no mechanisms to mirror the class hierarchy in the design. In such cases, the benefits of object-oriented design can only be partially realized, though even so, the notions of class and object, coupled with the shift in viewpoint, are valuable in themselves.

3. OBJECT-ORIENTED DESIGN

This section discusses object-oriented design in the context of the general concepts described previously. First, we concentrate on issues that, while not unique to the object-oriented approach, are of particular relevance. In general, this is due to facets of the object-oriented approach that highlight specific issues, or because of claims for the superiority of object-oriented approaches with regard to the issue at hand. This is followed by a discussion of approaches to object-oriented design that seek to address these issues and capitalize on the advantages of the metaphor.

3.1 Issues

3.1.1 Formal and Rigorous Development

An important issue is the means of specifying the behavior and defining the properties of objects in a class (or class hierarchy). One possibility is to use an informal description patterned after the process specifications (PSPECS) associated with structured analysis and design. However, many workers in the field believe there is much to be gained from using formal description notations such as Z [13, 43], , Obj [15], or Larch [17] to define the invariant state and applicable operations (methods) in a mathematically precise way. Indeed, there is even work on extending Z to Object-Z [14] in order to make the connection between specification and design even more explicit.

Another approach is to embed support for mathematical specification directly in a programming language. In this way, much of the design information is captured in the program text itself, thus minimizing the burden of keeping design documentation and related implementation consistent. Among the first languages to provide such support was Eiffel [30], where specific language clauses support the definition of class invariants (internal state consistency conditions), as well as method pre- and post-conditions. In Avalon/C++ [52], the use of Larch is complemented by the definition of a specific Larch interface language for C++. This in turn is used in the design and specification of objects in C++. Finally, the work done on A++ [9], combines the algebraic style of Larch and the model orientation of Eiffel to describe and reason about the properties of C++ classes.

What makes this work so interesting is that objects (or, more precisely, classes of objects) appear to provide a natural cluster for rigorous specifications of behavior. While it is still rare to see object-oriented designs presented using formal notations, the effects of organizing a model using the entities in the system appears to make such specifications easier. Hence, one candidate indicator of a high-quality design in an object-oriented technology would be the presence of a rigorous mathematical specification of each class's functional characteristics. In the long run, it may turn out that object-oriented technology will have facilitated the development and use of formal software engineering methods.

3.1.2 Encapsulation and Information Hiding

Flawed designs are undesirable, even if formally and precisely specified. This observation leads to consideration of other factors in the evaluation of design quality. Among the factors most often cited with respect to the object-oriented approach are encapsulation and information hiding, whose influence can be traced to the classic paper by Parnas [35].

Information hiding is the principle that each module in a system should maintain an *implementation secret*, such as the format of a data structure or the commands used to manipulate an I/O device. The remainder of the program can access and manipulate the information held in a module only via a small set of interface procedures and functions. Thus, the module's secret is encapsulated as part of the internal state, and is not directly manipulable by other portions of the system. In effect, these principles lead to software construction based on well-defined modules that are internally cohesive, yet which are loosely coupled to achieve the overall system objectives.

If one identifies the generic term *module* with the object-oriented notion of a *class*, and *interface procedures and functions* with *messages and methods*, the analogy is quite clear. Secrets are contained in each object's instance variables, which store the state information required to interpret and respond to each message. These variables are only manipulable in accordance with the specification of the object's behavior. However, the existence of multiple objects per class extends the basic concept of information hiding, in that there may be many instances of a class, each exhibiting similar behavior. For example, in a distributed system there might be many communications channels, each abiding by a common protocol. The secrets maintained for each channel (specific hardware interface to use, packet length restrictions, etc.) can be factored into a generic class definition. Channel creation reduces to creation of a new object in this class, which from that point on maintains the secrets for this specific channel. Overall behavior is thus separated from implementation details.

There are potential pitfalls, however, especially when inheritance comes into play. In most object-oriented systems, each class has unrestricted access to instance variables in its ancestor classes. This has proven to be both an asset and a liability. Direct access can improve efficiency, but makes it more difficult to verify correct overall system behavior. The problems inherent in even controlled breaking of encapsulation have led many to discourage the use of direct access in the name of correctness, reusability, and resilience to change [41]. Indeed, such restrictions are a key aspect of the Law of Demeter [27], which was developed as a heuristic for assessing design quality. In simplest terms, this "law" forbids the direct access by methods in a class to any state information in another class, even an ancestor. Following this law, in general, leads to high internal cohesion and low coupling, even in the face of inheritance.

3.1.3 Class Organization

A third important issue in object-oriented design is class organization. This naturally divides into three subproblems, class identification, class hierarchy design, and class client/supplier relationships.

The most important, and most difficult, issue in object-oriented design is the proper identification of the classes in a system. This process may be aided by parallel efforts to identify, specify, and catalog reusable generic classes for a particular domain (see Section 3.1.4). In addition, traditional concepts of information hiding, encapsulation, and module cohesion also provide important heuristics for evaluating the appropriateness of candidate classes. Still, the major emphasis in this area continues to be the principles for identifying candidate classes, and for selecting those which are useful in solving the problem at hand.

Once the basic classes have been identified, the next problem to be solved is that of creating the proper inheritance hierarchy. Here we implicitly assume a language with inheritance is being used for implementation. If this is not the case, then a flat class structure is almost certainly the most appropriate, as the process of simulating inheritance in a language that does not provide this feature tends towards unmanageably complex implementations.

A well-developed class hierarchy will typically contain abstract classes near the root and concrete classes at the leaves. Of course, there is a continuum from abstract to concrete, and interior classes in the hierarchy serve to both extend and implement methods which are simply specified at higher levels.

Finally, the client/provider relationships must be specified (this is similar to a structure chart in structured design or the *uses* relation described by Parnas [35]). In the case of object-oriented design, the connections are most frequently between classes. For instance, in the *CLOSED FIGURE* hierarchy shown in Figure 3, it is common to associate a position with each figure, and the natural class of such positions is a *POINT*. In this case, CLOSED FIGURE is a *client* of POINT, in that every such *CLOSED FIGURE* object will have a POINT object instance variable to hold the figure's position.

Note that the client/provider relationship is more resilient to change than that of subclass/superclass. The reason for this is that clients can only depend on the externally specified interface of the provider class, and are thus immune to changes in the internal structure or algorithms of the provider. This is manifestly not the case with subclasses, unless a strict adherence to encapsulation is maintained.

3.1.4 Abstraction and Reusable Classes

The final design issue we will address is that of abstraction and reusability. One of the promises held out by proponents of the object-oriented approach is that of increased reusability. However, this is only a potential benefit, and while the object-oriented approach may make identification of such components easier, it is all to easy to compromise reusability under the pressure of delivery deadlines. Currently there are efforts underway in many corporations to identify, specify, and catalog classes of objects of importance to the organization as a whole. This is the role of object-oriented domain analysis (OODA), and as such is beyond the scope of this work.

However, it is often true that reusable classes evolve in a bottom-up fashion, when groups working on a series of related projects begin to notice commonalities among the classes being developed. In its best form, this results in a process known as *refactoring*, where common state or methods are promoted to an abstract superclass, and the specific classes are then reimplemented in light of these commonalities. Work in this area is quite new, and support for semi-automated refactoring is a basic research problem [34]. Still, the ability to easily refactor and reconfigure class hierarchies is important for the support of both reusability and system evolution.

3.2 Design Approaches

The previous section presented some of the issues involved in the successful application of object-oriented design. This section discusses approaches to the use of this technology that have been proposed to deal with these issues.

3.2.1 Software Construction as Contracting

One of the most eloquent and intuitively appealing metaphors for object-oriented design is that of *software contracts*. As described by Betrand Meyer [30], contracts are binding obligations between objects (or, most commonly, between objects in two classes). The contract specifies both what *preconditions* must be met by the client object (such as permissible parameter values, etc.), and what service will be delivered by the provider object when the preconditions are met (usually given as a *postcondition* for a message/method). In addition, the provider object may be constrained by a *class invariant* that defines relations between internal state information (instance variables) necessary to guarantee correct service. An example invariant would be that the top of stack index for an array-based stack class must be between zero and the largest legal array index.

The contracting metaphor meshes well with the mathematically based concepts of formal specification, as well as the more informal concepts of information hiding and encapsulation. The contract defines *exactly* the obligations of each party, as well as the expected result when these obligations are met. Meyer pushes this idea even further: if either party to a contract fails to meet the obligations, then the other party is under no constraint. In particular, as discussed by Meyer, this means that failure of the client object to meet a precondition results in unpredictable behavior by the provider object.

The clear benefit of this approach is a reduction in the amount of error checking code within each method, as it is the client's responsibility to ensure the preconditions are met. On the other hand, this does lead inevitably to the need for rigorous inspection to ensure that all contractual obligations are met. Indeed, the exception mechanism of Eiffel is specifically designed to handle *only* failures to meet contractual obligations. The viewpoint here is that exceptions represent a significant deviation from specified behavior, and they serve simply to allow the software to place the overall system in a safe, consistent state.

The contracting concept is clearly attractive. It is clean, intuitive, and formalizable. Indeed, this notion will appear in other guises as we turn to the (few) extant methods proposed for designing object-oriented software.

3.2.2 Design Methods

While several methods for object-oriented design have been proposed, two approaches have been most influential:

- The research performed at Tektronix [54], and

- The work of Grady Booch at Rational [6].

This section will briefly discuss both of these efforts.

The work at Tektronix has produced a method called *responsibility-driven design*. One of the appeals of this technique is that the supporting technology is deceptively simple, requiring only cardboard index cards. These cards are used to record class names, responsibilities, and collaborators. Thus the name *CRC cards*. The goal is to take a specification, either formal or informal, and derive the appropriate classes in the system. As a first approximation, the nouns found in the specification are used as a first set of candidate classes. After weeding out duplicate and synonymous terms, the names of the classes are written at the top of individual index cards.

The next step is the identification of responsibilities, the information or state for which each class is responsible. At this point, a class may be split if the responsibilities are too many or too diffuse. Similarly, a class which has no responsibilities will be eliminated at this step (actually, put to one side, as later refinements may point up responsibilities not evident at this stage). At the end of this stage, each index card has been filled in with a set of responsibilities (typically 4 - 7).

Once responsibilities have been assigned, the next step is to identify each class's *collaborators*, that is, the other classes in the system on which the current class depends in order to fulfill its responsibilities. This, in essence, is where client/provider relationships are identified.

The final steps involve rearranging the classes into a hierarchy (i.e., identifying common responsibilities best factored into an abstract superclass), defining the specific interface to each class (that is, the messages the objects respond to and the results they return). In addition, for large systems another step is required to identify subsystems--classes of objects which, to a client, act as a cohesive unit. After this, the design is stable, and implementation can commence.

It must be stressed that the CRC method is *not* a linear progression as described above. Instead, as with most object-oriented methods, CRC is iterative, cycling through the stages until a stable design evolves. Indeed, this is one of the strengths of the approach, as it can be used continuously to evolve a software system to respond to changes in the environment where the system is employed.

By contrast, Booch's approach emphasizes the documentation of the design products, rather than a method for their development. In fact, though the book contains case studies in a variety of object-oriented and object-based languages, its value lies in the common notation it promotes for recording the final design. Indeed, it appears that the products of a CRC design effort could be easily transformed into Booch's notation almost mechanically. The benefits are a graphic design that can be manipulated on a workstation, rather than the collection of index cards that form the basis for a CRC design. In the long run, such machine readable versions of the design are easier to maintain and verify against the software that is developed.

3.2.3 Reusable Frameworks

One final theme of note is the emergence of reusable frameworks [53]. Frameworks are part design and part implementation. They are collection of classes that provide a skeleton software system which is embellished as necessary for a particular application. The most well-known framework is *model-view-controller* (or *MVC*) used for graphical interface development under Smalltalk-80 [23, 24]. Indeed, much of the research in framework development has focused on user interface issues, for example MacAPP for Apple's Object-Pascal [38] and ET++ for C++ and the X windowing system [51].

The concept behind frameworks reflects the observation that in many systems there is a core application that is unique, embedded in a context that is essentially constant between applications. As windowing interfaces are the most obvious common element, these have been the basis for most early frameworks, though seminal research in frameworks for other applications is beginning to be published [33]. However, in keeping with tradition, we will employ user interface frameworks, specifically MVC, as examples of the concept.

In simple terms, an application consists of one or more objects that contain the state information of the system. We will call these objects the *model*, as they model or simulate something of interest to the computer user. Normally the model is programmed to meet a specific function, but sometimes an existing class can be used as a prototype or exemplar. However, the displayed output is often stereotypical. To capture this commonality, MVC embodies a set of distinct graphical *views* that can be associated with a model. Each view captures a given aspect of the model of interest to the user. For example, if the model were a binary tree, one view might be a graph of the structure, while another view might present the same information as indented text.

A controller is an object that handles input events (keyboard and mouse activities) for a given view. Using the binary tree example again, one might rearrange the nodes in the tree by clicking and dragging the graphical nodes or by deleting and reinserting the lines in the textual display.

This uncovers the problem of communicating such changes to the model (so that it can update its internal version of the tree) and broadcasting such state changes to other views (for example, the graphical view if the tree is changed by editing the text view). MVC defines the protocol used to broadcast these changes, both from views to the model, and outward from the model to the views. Thus the framework consists of the following:

- A well-defined communications protocol between the model and its various views and controllers.
- A set of views and controllers which can be used as-is, or modified (by inheritance) to create special purpose behavior.
- An overall structure for applications that precisely defines the role of the application objects (model) within the MVC paradigm.

The greatest benefit accrues to frameworks when they provide a pattern of behavior and a library of classes to ease the development of such behavior. Thus frameworks provide both reusable code (in the library classes) as well as reusable design (in the overall application structure and inter-class communication protocol).

4. QUALITY METRICS for OBJECT-ORIENTED DEVELOPMENT

4.1 Software Quality Framework

The preceding sections have presented an overview of object-oriented concepts and current themes in object-oriented design. In this section, we concentrate on the use of metrics to evaluate software developed using object-oriented technology. It must be stressed that this technology is still rapidly evolving, and to date little attention has been focused on measurement of object-oriented artifacts. Thus, any metrics are tentative at best, and reflect primarily subjective value judgements by practitioners in the field. The appropriateness of the proposed measures must be continually evaluated, and must carefully track further developments in research and practice.

Nevertheless, the current state-of-the-art is mature enough to propose reasonable quality indicators. The following subsections consider the impact of object-oriented techniques within this context of the RL Software Quality Framework [7]. The last of these subsections contains a list and discussion of candidate questions related to object-oriented development that could be incorporated into the existing Framework.

The software quality framework is a hierarchy, proceeding from user-oriented quality factors through software-oriented quality criteria to measurable attributes embodied in metrics and metric elements (the actual questions used to score quality). The investigation of the effects of object-oriented technology on the framework proceeded in a top down fashion. The remainder of this section discusses the factors, criteria, and metrics that we believe would be significantly affected by the use of an object-oriented development technology.

4.2 Software Quality Factors

Of the thirteen factors, seven would be impacted, either strongly or moderately, by the use of object-oriented technology. The impacted factors are mostly concentrated in the Framework's Design and Adaptation factor categories, with only one impacted factor located in the Performance factor category. The following discussion focuses on each of these factors in turn, providing a sketch of the effects of object-oriented technology, and indicating the expected impact (strong or moderate).

4.2.1 **Usability** - How easily can the software be used?

Object-oriented technology, especially user interface prototyping packages, should help clarify usability issues earlier in the development cycle. In addition, adoption of standard object-oriented user interface packages should ease the transition from system to system.

Expected Impact: MODERATE

4.2.2 **Correctness** - How well does the software conform to its requirements?

The use of object-oriented technology, especially from high level design through implementation, provides a common language and notation that should help mitigate against mistakes at the transition between stages. In addition, object-oriented technology makes it easier to apply formal methods, in that the state of the system is subdivided along logical class boundaries.

Expected Impact: MODERATE

4.2.3 **Maintainability** - How easily can defects be repaired?

Object-oriented technology capitalizes on the concepts of encapsulation and information hiding to provide well-defined interaction boundaries. Coupled with a design process that explicitly identifies inter-class collaborations, the effect should be to enhance defect identification and elimination.

Expected Impact: STRONG

4.2.4 **Expandability** - How easily can the software be expanded or upgraded (i.e., how easily can the product *evolve*)?

Most proponents of object-oriented technology believe this is an area where the object-oriented approach is significantly superior to traditional functional design. One reason is that the objects or entities in a system change less frequently and dramatically than do the specific functions the system is expected to perform. By encapsulating functionality in objects, it becomes easier to identify the classes impacted by a proposed change, and often the effects of the change can be isolated to a few, well-defined objects. In addition, the inheritance can be used to specialize or extend the behavior of existing classes with little or no effect on existing object interaction.

Expected Impact: STRONG

4.2.5 **Flexibility** - How easy is it to change the software?

The advantages of object-oriented technology cited for the maintainability and expandability factors apply equally as well to flexibility.

Expected Impact: STRONG

4.2.6 **Interoperability** - How easy is it to interface the software with another system?

The use of the object-oriented paradigm supports the isolation of external interfaces in well-defined classes of interface objects. This makes it possible to abstract away the details of interacting with another hardware or software system, and in effect shielding the bulk of the current system from the complexities of interoperability. In addition, the adoption of standard reusable frameworks encourages development classes and objects that conform to standard interaction protocols, enhancing interoperability.

Expected Impact: MODERATE

4.2.7 **Reusability** - How easy is it to convert the software for use in another application?

Reusability is another area where proponents of object-oriented technology claim superiority to standard functional approaches. In particular, the use of object-oriented domain analysis (OODA) coupled with object-oriented requirements analysis (OORA) provides the opportunity to leverage off the work done in identifying classes of generic use. Even if code cannot be directly reused, design reuse can dramatically reduce both development time and residual errors.

Note that there are two aspects to reuse: the development of reusable components or designs as part of the current project, and the incorporation of reusable components or designs from previous projects. While the current Framework includes only the former, object-oriented technology, though concepts such as abstract classes and reusable frameworks, offers the promise of easier identification and incorporation of reusable components from the past.

Expected Impact: STRONG

4.3 Software Quality Criteria

Each factor is associated with one or more criteria, which indicate the extent to which a piece of software exhibits the factor. It is often the case that a given criterion will be associated with several factors. For example, the criterion of *generality* is important for expandability, flexibility, and reusability. Obviously, the criteria of interest were a subset of those affecting the factors previously identified. Of the 29 criteria used in the framework, the following 10 appear to be most relevant in assessing the use of object-oriented technology. Each criteria is followed by its standard Framework two character acronym.

4.3.1 Application Independence (AP) - Concerns independence of the software from particular operating systems, database systems, computer architecture, and microcode.

Abstract classes can be used to define prototypical behavior required by the operating system, database system, or underlying architecture. Concrete subclasses adhering to the abstract specification can then be developed for each specific system, isolating the bulk of the application from these details. This approach has already been used to advantage by the implementers of Smalltalk/V, where generic user interface classes are specialize for the different environments on which the product is supported.

4.3.2 Augmentability (AT) - Concerns the ability to expand the functionality or data in a system.

Additional or enhanced functionality can be achieved by the mechanisms of inheritance and dynamic binding of methods. New data objects can be introduced as either new classes or subclasses of existing classes.

4.3.3 Completeness (CP) - Concerns the provision of functionality providing full implementation of the requirements.

This is an area where the application of formal methods in conjunction with object-oriented technology can improve analysis and design.

4.3.4 Consistency (CS) - Concerns the provision of uniform design and implementation techniques and notation.

By focusing on the objects and object classes from the earliest stages of development, the progress becomes one of continual refinement of a common model. This is in contrast to many function-oriented techniques, where the notation and underlying concepts change radically from specification to design, and from design to implementation.

4.3.5 Functional Scope (FS) - Concerns the commonality of functions among applications.

The use of abstract superclasses supports the definition of reusable design components that support common functionality. Specific concrete subclasses can form the basis of reusable, common application components.

4.3.6 Generality (GE) - Concerns breadth of functionality performed with respect to an application.

The use of object-oriented domain analysis and object-oriented requirements analysis can lead to the identification of classes with general, useful behavior. With inheritance, these classes can be optimized or enhanced incrementally as needed by a specific application.

4.3.7 Modularity (MO) - Concerns the provision of a structure of cohesive components with optimal coupling.

Classes that are well specified and designed provide a natural, highly cohesive form of modularity. Indeed, Meyer goes so far as to identify the concepts of *class* and *module* [30]. Even if this extreme position is not adopted, the class provides a solid focal point for cohesive collections of data and functions (methods).

In addition, object coupling is easily identified and optimized when using the various manual (CRC) and graphical (Booch) notations. Finally, the class and object interaction information provides guidance as to the packaging of classes into coherent subsystems.

4.3.8 Operability (OP) - Concerns the operations and procedures for using the software that collects inputs and provides useful outputs.

This criterion has great influence on the usability of the resultant application. In particular, object-oriented interface packages and associated interface standards provide an excellent vehicle for prototyping interfaces and for ensuring interface consistency among related applications.

4.3.9 Simplicity (SI) - Concerns the definition and implementation of functions in the most noncomplex and understandable manner.

Proponents of the object-oriented approach argue that it is easier to explain and understand systems defined in terms of the component entities rather than the specific functions. In addition, the common vocabulary and notation possible at all stages enhances the overall system simplicity. Thus, the appropriate use of this technology should lead to systems whose structure is less complex, and whose components can be more easily understood.

4.3.10 System Clarity (ST) - Concerns the clear description of program structure in a non-complex and understandable manner.

This is similar to simplicity, except that the focus is on higher level structural issues. The object-oriented approach defines highly cohesive substructures, making it easier to understand the possible interactions in the application. In addition, as the objects and high-level connections tend to change relatively slowly, the clarity tends to remain high even in the face of ongoing evolution.

4.4 Metric Enhancement for Object-Oriented Development

Each criterion has associated with it one or more *metrics*. These metrics are quantitative measures that indicate the likelihood that the criterion is met by the product under consideration. The metrics, in turn, consist of *metric elements*. These elements are typically in question format and are used to quantize each metric, and by extension, the criterion and the factors it influences.

As in the case of quality factors and quality criteria, not all metrics are relevant when assessing software developed using object-oriented technology. Following are those metrics deemed important for object-oriented development, along with the specific metric elements being proposed for use with object-oriented technology. For each metric element, a short rationale is given for its inclusion in the list, and the metric worksheets where the element should appear are listed. Each new metric element is identified using the Framework's metric element format. Metric elements in each group are numbered sequentially, starting with the first available identifier. For example, the Framework currently includes one question in the AP.1 metric group, AP.1(1). The new metric elements shown below for group AP.1 are identified beginning with AP.1(2).

4.4.1 Application Independence

4.4.1.1 Database Management Implementation (AP.1)

Classes can and should be designed to isolate the body of the application from the particular details of the database system. The objects in these classes should reflect the logical view of the database required by the application, rather than the specific views supported by the database package itself.

AP.1(2) Are classes defined to provide abstract access to the database system? [Y/N/NA]

RATIONALE: Encapsulates knowledge about the details of the database system structure in a small number of related classes.

WORKSHEETS: 1, 2

AP.1(3) Are accesses to the database system via objects in the classes defined in AP.1(2)? [Y/N/NA]

RATIONALE: Helps insure consistent access to database information.

WORKSHEETS: 1, 2

AP.1(4) Do classes exist in the design (implementation) that provide abstract access to the database system? [Y/N/NA]

RATIONALE: Design and implementation version of AP.1(2)

WORKSHEETS: 3A, 4A

AP.1(5) Does this unit perform all database accesses via objects in classes defined for this purpose? [Y/N/NA]?

RATIONALE: This is a detailed design and implementation version (unit level) of the general question AP.1(3).

WORKSHEETS: 3B, 4B

AP.1(6) (a). Number of units accessing the database.

(b). Number of units using objects in the provided classes.

(c). Enter (b) / (a)

RATIONALE: This is a detailed design and implementation version (CSCI level) of the general question AP.1(3).

WORKSHEETS: 3A, 4A

4.4.1.2 Data Structures (AP.2)

In object-oriented technology, all information on data structure is maintained in the objects themselves. Thus issues related to the naming and use of data structures are intimately tied to the class structure and object interactions.

AP.2(5) Is there a standard for identifying all global objects and their associated classes used by a unit? [Y/N/NA]

RATIONALE: Identifies potential dependencies on particular global entities.

WORKSHEETS: 0, 1, 2

AP.2(6) Is there a standard for identifying the class of all parameter objects and local objects used in a unit? [Y/N/NA]

RATIONALE: Identifies all dependencies on classes that will be required should the unit be used in another application.

WORKSHEETS: 0, 1, 2

AP.2(7). Does the unit follow the standard in identifying all global objects and their associated classes used by a unit? [Y/N/NA]

RATIONALE: This is the detailed design and implementation version (unit level) of question AP.2(5)

WORKSHEETS: 3B, 4B

AP.2(8) (a) Number of units in this CSCI

(b) Number of units following the standards for identifying global objects

(c) Enter (b) / (a)

RATIONALE: This is a detailed design and implementation version (CSCI level) of the general question AP.2.(5)

WORKSHEETS: 3A, 4A

AP.2(9) Does the unit follow the standard for identifying the class of all parameter objects and local objects used in a unit? [Y/N/NA]

RATIONALE: This is the detailed design and implementation version (unit level) of question AP.2(6).

WORKSHEETS: 3B, 4B

AP.2(10) (a) Number of units in this CSCI:

(b) Number of units following the standards for identifying parameter and local object classes:

(c) Enter (b) / (a)

RATIONALE: This is a detailed design and implementation version (CSCI level) of the general question AP.2(6)

WORKSHEETS: 3A, 4A

4.4.2 Augmentability

In object-oriented systems, design extensibility primarily involves the development or reuse of abstract classes and reusable frameworks providing an open-ended path for systematic software evolution.

4.4.2.1 Design Extensibility (AT.4)

AT.4(4) Is documentation available describing domain analysis for generic classes of relevance to the application? [Y/N/NA]

RATIONALE: Existing class definitions for the application domain help to insure that general, extensible classes are used for high level architectural decisions.

WORKSHEETS: 0, 1

AT.4(5) Is documentation available describing feasibility studies done with the goal of identifying reusable frameworks for incorporation in the application? [Y/N/NA]

RATIONALE: Frameworks are by their nature skeleton applications designed with extensibility and evolution in mind.

WORKSHEETS: 0, 1

AT.4(6) Is there documentation discussing proposed classes in terms of their potential for extension? [Y/N/NA]

RATIONALE: Classes are the primary unit of extensibility and the potential for extension is a key aspect of high-level design.

WORKSHEETS: 2

AT.4(7) Is there documentation of classes in the system designed specifically to address expected areas of extension? [Y/N/NA]

RATIONALE: Classes which encapsulate data and functions that are candidates for anticipated extension should be identified as such. In part, this is a design oriented version of question AT.4(4).

WORKSHEETS: 2

AT.4(8) Is there documentation describing the use of frameworks as part of the skeleton design? [Y/N/NA]

RATIONALE: This is the high-level design version of question AT.4(5)

WORKSHEET: 2

4.4.3 Completeness

The use of formal methods to describe the state invariants of the objects in each class, as well as the preconditions and postconditions for the use of each message/method will help insure completeness and correctness. In addition, some items on the checklist should be phrased in terms appropriate for object-oriented technology.

4.4.3.1 Completeness Checklist (CP.1)

CP.1(12) Has every required function been assigned as a responsibility of an appropriate class? [Y/N/NA]

RATIONALE: This insures that all required functionality has been addressed.

WORKSHEETS: 1, 2, 3A

CP.1(13) Has every class been assigned a clear set of responsibilities? [Y/N/NA]

RATIONALE: Guarantees that each class has an appropriate role in the overall system.

WORKSHEETS: 1, 2, 3A

CP.1(14) Are each class's collaborating classes identified and are the collaborations defined? [Y/N/NA]

RATIONALE: Identifies the inter-object communication within the application.

WORKSHEETS: 1, 2, 3A

CP.1(15) In the class hierarchy, is each subclass's behavior defined relative to that of the parent class? [Y/N/NA]

RATIONALE: Identifies the role of inheritance within the overall application.

WORKSHEETS: 1, 2, 3A

CP.1(16) Do class descriptions include a precise (e.g., mathematical) description of the state invariant for each object in the class, as well as pre and post conditions for the use of each method? [Y/N/NA]

RATIONALE: Formal specifications document class behavior precisely, and can uncover ambiguities and design omissions.

WORKSHEETS: 1, 2

CP.1(17) Do the classes in this unit have precise (e.g., mathematical) descriptions of the state invariant for each object, as well as pre and post conditions for the use of each method? [Y/N/NA]

RATIONALE: Detailed design version (unit level) of question CP.1(16)

WORKSHEETS: 3B

CP.1(18) (a) Number of units in this CSCI:

(b) Number of units with precise descriptions of class behavior

(c) Enter (b) / (a)

RATIONALE: Detailed design version (CSCI level) of question CP.1(16)

WORKSHEETS: 3A

4.4.4 Consistency

NOTE: One of the primary facets of object-oriented technology is the intimate coupling of data (state) and procedure (methods) within the framework of a class. For this reason, the functional division into procedure consistency and data consistency is inappropriate, and the questions for consistency will be in terms of a new combined metric category, *Object Consistency*.

4.4.4.1 Object Consistency (CS.3)

CS.3(1) Has a specific standard been established for documenting class responsibilities? [Y/N/NA]

RATIONALE: The responsibilities of each class (what behavior it must exhibit) are important for evaluating designs, and a standard way of representing these is essential.

WORKSHEETS: 0, 1

CS.3(2) Has a specific standard been established for representing class relationships (e.g., Booch diagrams, CRC cards)? [Y/N/NA]

RATIONALE: A standard way of representing both the inheritance hierarchy and the client/provider relationships is essential to effective communication.

WORKSHEETS: 0, 1

CS.3(3) Has a specific standard been established for naming both classes and their methods? [Y/N/NA]

RATIONALE: Naming conventions are crucial in order to avoid name clashes and ambiguities.

WORKSHEETS: 0, 1

CS.3(4) Has a specific standard been established for naming all global objects? [Y/N/NA]

RATIONALE: It is important to be able to distinguish global from local references when evaluating and maintaining software.

WORKSHEETS: 0, 1

CS.3(5) Does the design follow standards for documenting class responsibilities? [Y/N/NA]

RATIONALE: Preliminary design version of question CS.3(1)

WORKSHEETS: 2

CS.3(6) Does the design follow the standard for representing class relationships? [Y/N/NA]

RATIONALE: Preliminary design version of question CS.3(2).

WORKSHEETS: 2

CS.3(7) Does the design follow the standard for naming classes and methods? [Y/N/NA]

RATIONALE: Preliminary design version of question CS.3(3).

WORKSHEETS: 2

CS.3(8) Does the design follow the standard for naming all global objects? [Y/N/NA]

RATIONALE: Preliminary design version of question CS.3(4)

WORKSHEETS: 2

CS.3(9) Does the unit follow standards for documenting class responsibilities? [Y/N/NA]

RATIONALE: Unit level version of question CS.3(1).

WORKSHEETS: 3B

CS.3(10) (a) Number of units in this CSCI:
(b) Number of units following standards for class responsibilities:
(c) Enter (b) / (a)

RATIONALE: CSCI level version of question CS.3(1).

WORKSHEETS: 3A

CS.3(11) Does unit follow the standards for representing class relationships? [Y/N/NA]

RATIONALE: Unit level version of question CS.3(2)

WORKSHEETS: 3B

CS.3(12) (a) Number of units in this CSCI:
(b) Number of units following standards for class relationships:
(c) Enter (b) / (a):

RATIONALE: CSCI level version of question CS.3(2).

WORKSHEETS: 3A

CS.3(13) Does the unit follow the standard for naming classes and methods? [Y/N/NA]

RATIONALE: Unit level version of question CS.3(3).

WORKSHEETS: 3B

CS.3(14). (a) Number of units in this CSCI:
(b) Number of units following standards for class and method naming
(c) Enter (b) / (a):

RATIONALE: CSCI level version of question CS.3(3)

WORKSHEETS: 3A

CS.3(15) Does the unit follow the standard for naming all global objects? [Y/N/NA]

RATIONALE: Unit version of question CS.3(4)

WORKSHEETS: 3B

CS.3(16) (a) Number of units in this CSCI:
(b) Number of units following standards for naming global objects:
(c) Enter (b) / (a):

RATIONALE: CSCI level version of question CS.3(4).

WORKSHEETS: 3A

4.4.5 Functional Scope (FS)

4.4.5.1 Functional Commonality (FS.2)

This metric is concerned with the commonality of function with other similar applications. Our metric elements interpret the concept of "function" liberally as a synonym for class and object.

FS.2(7) Are there requirements to construct abstract super classes so as to enhance design reusability in similar applications?

RATIONALE: Abstract classes promote commonality and reuse.

WORKSHEETS: 0, 1

FS.2(8) Are there requirements to construct concrete classes so as to facilitate their use in similar applications.

RATIONALE: Such construction supports code commonality as well as design commonality.

WORKSHEETS 0, 1

FS.2(9)
- (a) How many classes are in the system?
- (b) How many classes are likely to satisfy requirements of similar applications?
- (c) Enter (b) / (a)

RATIONALE: This element measures the actual commonality achieved by the design.

WORKSHEETS: 2

4.4.6 Generality (GE)

4.4.6.1 Unit Referencing

This measures generality by how many references there are to an entity (in our case, classes).

GE.1(2)
- (a) How many total superclasses?
- (b) How many superclasses ancestors of more than one subclass?
- (c) Enter (b) / (a)

RATIONALE: This measures the degree to which superclasses provide generally useful services to be extended.

WORKSHEETS: 2

GE.1(3)
- (a) How many total classes?
- (b) How many classes provide service to more than one client class?
- (c) Enter (b) / (a)

RATIONALE: This measures the degree to which a classes provide generally useful services within the application.

WORKSHEETS: 3A, 4A

NOTE: Modularity using object-oriented technology is evaluated quite differently from that used in function-oriented structured design. In particular, the class (and its objects) are the focus rather than individual procedures. **Thus, existing worksheet questions related to structured design concepts are not applicable to software**

developments using object-oriented technology and would be replaced by their equivalents in the metric element lists below.

4.4.7 Modularity (MO)

4.4.7.1 Modular Implementation (MO.1)

MO.1(10) Are there requirements to develop all classes using an established object-oriented technique (e.g., Booch or CRC methods)? [Y/N/NA]

RATIONALE: Replaces MO.1(1)

WORKSHEETS: 0, 1

MO.1(11) Are all classes developed according to an established object-oriented technique? [Y/N/NA]

RATIONALE: Replaces MO.1(2)

WORKSHEETS: 0, 1, 2, 3A, 4A

MO.1(12) Do any of the methods in the current class have implementations exceeding 50 non-comment source lines? [Y/N/NA]

RATIONALE: Replaces MO.1(3) at unit level. In general, method bodies should be very small.

WORKSHEETS: 3B, 4B

MO.1(13) (a) How many total classes ?
(b) How many classes have methods with over 50 non-comment source lines?
(c) Enter 1 - ((b) / (a))

RATIONALE: Replaces MO.1(3) at CSCI level.

WORKSHEETS: 3A, 4A

MO.1(14) Do each of the methods in the current class have a single processing objective? [Y/N/NA]

RATIONALE: Replaces MO.1(9) at unit level.

WORKSHEETS: 3B, 4B

MO.1(15) (a) How many total classes?

(b) How many classes have only methods with a single processing objective?

(c) Enter (b) / (a)

RATIONALE: Replaces MO.1(9) at CSCI level.

WORKSHEETS 3A, 4A

4.4.7.2 Modular Design (MO.2)

Modular design in object-oriented systems is evaluated more on the relationships between classes than on a simple function-oriented basis. These metrics reflect this viewpoint.

MO.2(6) Are there requirements limiting the number of collaborations any given class may engage in? [Y/N/NA]

RATIONALE: Increases in collaborations correlate to increased object coupling.

WORKSHEETS: 0, 1

MO.2(7) Do the classes in this CSCI obey the limitations on the number of collaborating classes? [Y/N/NA]

RATIONALE: Design version of MO.2(6)

WORKSHEETS: 2, 3A

MO.2(8) Are there requirements limiting the direct access to instance variables in a superclass? [Y/N/NA]

RATIONALE: Such access violates encapsulation, and makes it more difficult to verify the correct operation of a class. It creates a form of common coupling between the two classes.

WORKSHEETS: 0

MO.2(9) (a) How many classes in this CSCI?

(b) How many classes directly access ancestor class instance variables?

(c) Enter (b) / (a)

RATIONALE: Design version of MO.2(8)

WORKSHEETS: 2, 3A, 4A

4.4.8 Operability (OP)

NOTE: The next two metrics relate to the use of object-oriented user interface packages and general interface frameworks. They are grouped into a new metric category, *user communicativeness* (OP.4). The same questions apply to both input and output communicativeness.

4.4.8.1 User Communicativeness (OP.4)

OP.4(1) Has an object oriented interface package been used to prototype prospective user interfaces? [Y/N/NA]

RATIONALE: Prototyping is important in establishing the appropriate interface.

WORKSHEETS: 0, 1

OP.4(2) Has an engineering study been performed on the feasibility of employing a reusable framework as the basis of the user interface?

RATIONALE: Where appropriate, reusable frameworks help establish common look and feel among applications.

WORKSHEETS: 0, 1

4.4.9 Simplicity (SI)

4.4.9.1 Design Structure (SI.1)

These elements reflect the object-oriented version of many design simplicity elements originally developed for functional decomposition.

SI.1(11) Are there appropriate graphical representations of the class hierarchy and class collaborations (e.g., Booch diagrams)? [Y/N/NA]

RATIONALE: These diagrams provide a high level view of overall application architecture.

WORKSHEETS: 0, 1

SI.1(12) Have the graphical representations of class hierarchy and class collaborations been followed? [Y/N/NA]

RATIONALE: Design version of SI.1(11)

WORKSHEETS: 2, 3A

SI.1(13) Are there programming standards for the development of classes and methods? [Y/N/NA]

RATIONALE: Similar to current SI.1(8)

WORKSHEETS: 0, 1, 2

SI.1(14) Do the classes in this unit follow the defined programming standards? [Y/N/NA]

RATIONALE: Detailed design and implementation version of SI.1(13) (unit level)

WORKSHEETS: 3B, 4B

SI.1(15) (a) How many units in this CSCI?
(b) How many units follow the programming standards?
(c) Enter (b) / (a)

RATIONALE: Detailed design and implementation version of SI.1(14) (CSCI level)

WORKSHEETS: 3A, 4A

4.4.10 System Clarity (ST)

4.4.10.1 Interface Complexity (ST.1)

Interface complexity relates to both the inheritance structure, and the form, number, and specification of methods.

ST.1(7) Is each method clearly and precisely specified (e.g., via pre and post conditions and its effect on the object state). [Y/N/NA]

RATIONALE: Such specifications are essential to understanding a class's interface.

WORKSHEETS: 2, 3A

ST.1(8) Are methods categorized as to whether they interrogate the object state or alter this state? [Y/N/NA]

RATIONALE: Such separation makes it easier to determine the effects of sending a message to the object.

WORKSHEETS: 2, 3A

ST.1(9) Is the application inheritance structure fewer than five levels deep? [Y/N/NA]

RATIONALE: Structures exceeding this limit make it more difficult to determine the precise method that may be used when and object is sent a message.

WORKSHEETS: 2, 3A

ST.1(10) Is each class free from methods with duplicated or overlapping functionality? [Y/N/NA]

RATIONALE: Such duplication increases the complexity of the class, and can lead to confusion as to which message should be used.

WORKSHEETS: 2, 3A

5. REFERENCES

[1] G. Agha and C. Hewitt, "Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming," in B. Shriver and P. Wegner, (editors), *Research Directions in Object-Oriented Programming*, MIT Press, Cambridge, MA, 1987.

[2] P. America and F. van der Linden, "A Parallel Object-Oriented Language with Inheritance and Subtyping," *OOPSLA/ECOOP '90*, pp. 161-168, Ottawa, Canada, October,1990.

[3] D. Bobrow, L. DeMichiel, R. Gabriel, G. Kiczales, D. Moon, and S. Keene, *The Common Lisp Object System*, Technical Report 88-002R, X3J13 Standards Committee, 1988.

[4] G. Booch, *Software Components with Ada*, Benjamin/Cummings, Reading, MA, 1987.

[5] G.Booch, "Design of the C++ Booch Components," *OOPSLA/ECOOP '90*, pp. 1-11, Ottawa, Canada, October 1990.

[6] G. Booch, *Object Oriented Design with Applications*, Benjamin/Cummings, Reading, MA, 1990.

[7] T. Bowen, G. Wigle, and J. Tsai, *Specification of Software Quality Attributes*, Technical Report RADC-TR-85-37, Rome Air Development Center, 1985.

[8] C. Chee, C. Ng, and M. Sim, "Towards an Object-Oriented Analysis and Design Methodology (TOAD)," *Symposium on Object-Oriented Programming Emphasizing Practical Applications*, pp. 1-15, Marist College, September 1990.

[9] M. Cline and D. Lea, "The behavior of C++ Classes," *Symposium on Object-Oriented Programming Emphasizing Practical Applications*, pp. 81-91, Marist College, September 1990.

[10] P. Coad and E. Yourdon, *Object-Oriented Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[11] B. Cox, *Object-Oriented Programming*, Addison-Wesley, Reading, MA, 1986.

[12] O.J. Dahl, B. Myhrhaug, and K. Nygaard, *The Simula 67 Common Base Language*, Norwegian Computing Centre, Oslo, Norway, 1968.

[13] A. Diller, *Z: An Introduction to Formal Methods*, John Wiley and Sons, 1990.

[14] R. Duke, P. King, G. Rose, and G. Smith, *The Object-Z Specification Language Version 1.*, Technical Report 91-1, The University of Queensland, 1991.

[15] J. Goguen and J. Meseguer, "Unifying Functional, Object-Oriented, and Relational Programming with Logical Semantics," in B. Shriver and P. Wegner, (editors), *Research Directions in Object-Oriented Programming*, MIT Press, Cambridge, MA, 1987.

[16] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison Wesley, Reading, MA, 1983.

[17] J. Guttag, J. Horning, and A. Modet, *Report on the Larch Shared Language: Version 2.3*, Technical Report 58, DEC Systems Research Center, 1990.

[18] B. Henderson-Sellers and J. Edwards, "The Object-Oriented Systems Life Cycle," *Communications of the ACM*, (33,9), pp.142-159, September, 1990.

[19] C. A. R. Hoare, *Notes on Data Structuring in Structured Programming*, Academic Press, New York, 1972.

[20] D. Ingalls, "Design Principles Behind Smalltalk," *Byte*, (6,8), August,1981.

[21] B. Kernighan and D. Ritchie, *The C Programming Language 2nd ed*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[22] T. Korson and J. McGregor, "Understanding Object-Oriented: A Unifying Paradigm," *Communications of the ACM*, (33,9), pp. 41-60, September, 1990.

[23] W. LaLonde and J. Pugh, *Inside Smalltalk: Volume I*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[24] W. LaLonde and J. Pugh, Inside Smalltalk: Volume II, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[25] B. Lampson, J. Horning, R. London, J. Mitchell, and G. Popek, "Report on the Programming Language Euclid," *SIGPLAN Notices*, (12,2), pp. 1-79, February, 1977.

[26] M. Lehman and L. Belady, *Program Evolution: Processes of Change*, Academic Press, London, 1985.

[27] K. Lieberherr and I. Holland, "Assuring Good Style for Object-Oriented Programming," *IEEE Software*, (6,5), pp. 38-48, September, 1989.

[28] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. Schaffert, R. Scheifler, and A. Snyder, *CLU Reference Manual*, Springer-Verlag, New York, 1981.

[29] M. D. McIlroy, "Mass-Produced Software Components," in J. Buxton, P. Naur, and B. Randell, (editors), *Software Engineering Concepts and Techniques (1968 NATO Conference on Software Engineering)*, pp. 88-98. Van Nostrand Reinhold, 1976.

[30] B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, NY, 1988.

[31] B. Meyer, "Lessons From the Design of the Eiffel Libraries," *Communications of the ACM*, (33,9), pp. 69-88, September, 1990.

[32] D. Moon, "Object-Oriented Programming with Flavors," *OOPSLA '86*, pp. 1-8, Portland, OR, September, 1986.

[33] W. Opdyke and R. Johnson, "Refactoring: An Aid in Designing Application Frameworks," Symposium on Object-Oriented Programming Emphasizing Practical Applications, pp. 145-161. Marist College, September 1990.

[34] W. Opdyke and R. Johnson, "Refactoring: An Aid in Designing Application Frameworks," Symposium on Object-Oriented Programming Emphasizing Practical Applications, pp. 145-161. Marist College, September 1990.

[35] D. Parnas, "On the Criteria to be Used in Decomposing Systems Into Modules," *Communications of the ACM*, (15,12), pp. 1053-1058, December, 1972.

[36] W. Pun and R. Winder, "A Design Method for Object-Oriented Programming," *ECOOP '89* , pp. 225-242, University of Nottingham, July,1989.

[37] M. Sakkinen, "Disciplined Inheritance," *ECOOP'89*, pp. 39-56, University of Nottingham, July, 1989.

[38] K. Schmucker, *Object-Oriented Programming for the Macintosh*, Hayden Book Company, Hasbrouck Heights, NJ, 1986.

[39] S. Shlaer and S. Mellor, *Object-Oriented Systems Analysis*, Prentice-Hall, Englewood Cliffs, NY, 1988.

[40] B. Shriver and P. Wegner (editors), *Research Directions in Object Oriented Programming*, MIT Press, Cambridge, MA, 1987.

[41] A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," *OOPSLA '86*, pp. 38-45, Portland, OR, September 1986.

[42] *Classic Ada*, Software Productivity Solutions, Inc., Melbourne, FL.

[43] J. Spivey, *The Z Notation: A Reference Manual*, Prentice-Hall, 1989.

[44] G. Steele, *LISP*, Digital Press, Burlington, MA, 1984.

[45] L. Stein, "Delegation is Inheritance," *OOPSLA '87*, pp. 138-146, Orlando, FL, October, 1987.

[46] C. Strachey, "Fundamental Concepts in Programming Languages," *Lecture Notes for International Summer School in Computer Programming*, Copenhagen, August, 1967.

[47] B. Stroustrop, *The C++ Programming Language*, Addison Wesley, Reading, MA, 1986.

[48] U. S. Department of Defense, *Reference Manual for the Ada Programming Language*, 1983.

[49] D. Ungar, "Self: The power of Simplicity," *OOPSLA '87*, pp. 138-146, Orlando, FL, October,1987.

[50] P. Wegner, "Dimensions of Object-Based Language Design," *OOPSLA '87*, pp. 168-182, Orlando, FL, October 1987.

[51] A. Weinand, E. Gamma, and R. Marty, "ET++ - An Object Oriented Application Framework in C++," *OOPSLA '88*, pp. 46-57, San Diego, September, 1988.

[52] J. Wing, "Using Larch to Specify Avalon/C++ Objects," *IEEE Transactions on Software Engineering*, (16,9), pp. 1076-1088, September, 1990.

[53] R. Wirfs-Brock and R. Johnson, "Surveying Current Research in Object-Oriented Design," *Communications of the ACM*, (33,9), pp. 104-124, September, 1990.

[54] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[55] N. Wirth, "The programming language Pascal," *Acta Informatica* (1,1), pp. 35-63, 1971.

[56] N. Wirth, *Programming in Modula-2* (3rd edition), Springer-Verlag, New York, 1985.